Composing Data & Process Descriptions in the Design of Software Systems

by

Daniel Jackson

May 1988

This report is a minor revision of a thesis of the same title submitted to the Department of Electrical Engineering and Computer Science on May 6th, 1988 in partial fulfillment of the requirements for the degree of Master of Science.

Thesis Supervisor: Professor John V. Guttag

This research was supported primarily by an ITT International Fellowship, and in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-83-K-0125, by the National Science Foundation under grant 8706652-CCR, and by the NYNEX Corporation.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

This empty page was substituted for a blank page in the original document.

Abstract

Two paradigms are dominant in software development, the data paradigm and the process paradigm. Our contention is that relying exclusively on either is counter-productive.

In the data paradigm, a system is specified as operations acting on states. The process paradigm focuses on sequences of events. By analyzing two specifications for the same system, one in each paradigm, we show that the prime concerns of the approaches based on the data and process paradigms are *state* and *sequencing* respectively. Without explicit data, a system cannot take advantage of representation independence, a prerequisite of modularity; without sequencing notions, the components of a system cannot be connected in an abstract fashion. Fortunately, the qualities of the two paradigms are complementary, suggesting an approach that combines the two.

We present a framework in which data and process specifications are combined formally. A series of small examples shows how data and process specifications can be combined to build systems. We define formally what it means for an implementation to satisfy a specification, and we show a CLU implementation of one of our examples. Finally we outline a prescriptive notion of implementation in which the specification dictates the internal structure of the implementation.

Acknowledgements

I thank my supervisor, John Guttag, for his careful reading of many drafts. His perceptive suggestions have improved my presentation enormously. He has often saved me from worthless philosophizing, and has taught me the importance of convincing examples.

Alan Fekete explained CSP to me, and our discussions (and his astute comments on the drafts he kindly offered to read) helped me clarify my intuitions. My father, Michael Jackson, took my ideas much more seriously than they deserved; he suggested the connective I finally adopted, and his lucid and novel ideas about method have influenced by work throughout. Tobias Nipkow tolerated constant interruptions of his work and found several errors in the final draft. Jim Woodcock showed me an inspiring example of a specification using CSP and Z last year.

Thank you to Claudia Marbach for her insights into ticket machines, and for taking me out for sushi. Finally, I thank my parents for their encouragement, and for never complaining once about the cost of trans-Atlantic phone calls.

Two are better than one, for they have a greater benefit from their earnings.

Ecclesiastes, 4:9

Contents

List of Figures

Chapter 1

Introduction

Since the term 'software engineering' was coined in 1969, a multitude of languages, tools and techniques have been developed. Assessing different approaches is difficult; they address themselves to different stages in software development, and to different applications. We believe, however, that, whatever the approach, formal specifications are essential to systematic development of software. They avoid many of the pitfalls of natural language specifications, such as ambiguity and imprecision [Meyer 85]. Proofs of program correctness are only viable when the specifications are formal [Jones 85]. Some formal specifications are amenable to automatic consistency checks [Guttag 80], and the operational approaches [Zave 84], which advocate deriving programs mechanically by transformation, depend on them.

Paradoxically, the approaches based on formal specifications tend to be particularly hard to analyse. We suggest two reasons for this. First, the abstract mathematical nature of formal specification languages makes them suitable for a wide variety of applications, and their inventors are often reluctant to associate more specialized techniques with their languages. Secondly, stating the informal assumptions behind a specification language may be seen as undermining its objective, formal interpretation. We view this as detrimental to the progress of software development. The techniques accompanying formal specifications are not incidental; a specification language that lacks the backing of a firm technique is of little use. Method is crucial to software development, and we should not be inhibited from discussing it just because we cannot phrase our discussion in formal terms.

Some early approaches to software development divided programs into two parts, the data structures and the algorithms that manipulated them[1]. The advent of *data abstraction* refined the notion of data, separating the representation of data from its behaviour. Data was no longer viewed as passive; we began to define a data object in terms of the behaviour of the operations it offered, instead of the set of concrete values it could assume. This insight blurred the distinction between data and process, and it became fashionable to think of a system's data as central, and the process aspects as subsidiary [Liskov 86]. (The object-oriented approach [Booch 86] is perhaps the most

[1] For example, programs written in COBOL have a 'data division' in which the data structures are defined, and a 'procedure division' for the code itself. Wirth's book on program design is entitled *Algorithms + Data Structures = Programs* [Wirth 76].

extreme example of this view.) The process aspect has taken longer to mature. A corresponding insight has relegated data to an inferior position: the most abstract view of a system's behaviour considers only the sequences of events it engages in, and data is unnecessary. (This view is well represented by CSP [Hoare 85]).

The field has thus split into two schools: the data school, which advocates the primacy of data abstractions, and the process school which takes the opposite view. Examples of approaches in the data school are Larch [Guttag 85b], VDM [Jones 86] and Z [Hayes 86]; approaches of the process school include JSD [Jackson 83], PAISLey [Zave 82] and CSP [Hoare 85].

No practical approach can rely purely on data or process notions. We find that each approach adopts some elements from the competing school. However, the approaches are marred by their biases. The data approaches do not handle concurrency well, and they offer only crude composition mechanisms for constructing systems. The lack of data abstraction in the process approaches creates complexity and inflexibility to changes in data representation.

We contend that each school has much to offer, and that an effective approach to software development must contain ingredients from both schools, data and process, in a reasonable balance. This thesis has two goals: to demonstrate that an approach that combines data and process notions is feasible, and to clarify the differences between the schools, in the hope that this will lead to a better appreciation of the strengths and weaknesses of existing approaches.

Because we believe that the nature of the specifications used characterizes an approach, the thesis focuses on specification issues. We do not, however, consider how our specifications would be constructed methodically, nor do we give a practical scheme for implementation. These questions must be left to future research.

In Chapter 2, we consider data and process as paradigms for specification, basing our discussion on a small example. We give two specifications of a banking system, one in the process paradigm, the other in the data paradigm. We point out which aspects are treated well by each paradigm, and we discuss some of the common assumptions that advocates of the two paradigms make about their use and interpretation. By considering some simple elaborations of the system's requirements, we show that each paradigm handles some changes well and others badly.

Chapter 3 is the core of the thesis. We show a natural division of a specification into two parts, a process part and a data part. The meaning of the combined specification is given by a trace semantics in which each part denotes an 'object' that resembles a CSP process. The objects are composed with a parallel connective similar to that of CSP. The rest of the chapter gives some small examples that suggest how the approach may be used to structure systems.

Chapter 4 takes one of the specifications from Chapter 3, in a slightly elaborated form, and gives an implementation in CLU [Liskov 81]. We discuss briefly what issues would arise in developing an implementation technique that would be general and practical.

Chapter 5 surveys related work briefly and proposes some directions for future work. We outline a prescriptive notion of satisfaction, in which the specification dictates the structure, as well as the behaviour, of the implementation. We also explain why an earlier attempt of ours to combine data and process failed.

Chapter 2

Data & Process Paradigms

2.1 The Nature of the Difference

The distinction between the process and data paradigms is not a formal one; the approaches that typify the two paradigms define the notions of data and process in subtly different ways. Thus, for VDM, a data type is given by domain equations on predefined types, but for Larch, a data type is a based on an algebraically defined 'sort'. A CSP process (in its simplest incarnation) is an alphabet and a trace set, but a JSD process is a regular grammar of messages on a stream.

Most approaches incorporate both paradigms to some degree. A common assumption of data approaches, for example, is that operations are never called when their preconditions are false, and this sequencing constraint lends a process flavour to a data specification. Similarly, Hoare, in his exposition of CSP, presents examples in which the process names are indexed by the values of common types (integers, sets and sequences). In all existing approaches, though, one paradigm dominates, and the other is subsidiary.

The process and data paradigms are equally expressive in the formal sense – their languages are rich enough to encode all Turing machines. Our concerns are not mathematical but methodological. We do not judge an approach to specification by the richness of the mathematical structures it embodies. Instead, we care whether the approach helps us develop correct and flexible systems with confidence and efficiency. These are elusive goals, and there is little agreement about how they may be accomplished. However, we shall argue that each paradigm is only suitable for developing some aspects of a system, and has damaging consequences when applied elsewhere. Fortunately, the qualities of the paradigms are complementary, suggesting that a combination would be desirable. In Chapter 3 we define a language that divides a specification into two parts, data and process, and we give a framework in which to combine them.

2.2 Analysis

Our analysis of the two paradigms is based on a simple example. We describe a tiny banking system informally, and we then construct two formal specifications, one in each paradigm. The languages we use for the two specifications have been chosen to make the

presentation straightforward. The data specification is in the style of VDM [Jones 86], and the process specification is in CSP [Hoare 85][1].

2.2.1 A Banking Example

The bank administers accounts for its customers. An account is opened with an initial deposit. Thereafter, the customer may withdraw from the account any amount not greater than the balance, deposit funds in the account and enquire about the balance of the account. An account may be closed at any time. After it has been closed, no more transactions are permitted.

We can view the history of the banking system as a sequence of events. The behaviour of the system is given by the set of all possible histories. Each history, or *trace*, describes what happened in the system up to some moment, and is thus finite. However, there may be no bound on the length of the traces; if the banking system runs for ever, the trace set will be infinite. Here are some of the traces:

<>
<open-#1-$100, close-#1>
<open-#1-$100, payin-#1-$50, open-#2-$10, bal-#1-$150>
<open-#1-$50, open-#2-$70, close-#1, wdraw-#2-$50>

The first trace is empty; any system can do nothing. In the traces that follow, each event is a single banking transaction. Each event name has one or more parts. The first part is the kind of transaction: an opening, a closing, a withdrawal, a deposit or a balance enquiry. The second part indicates the account that is the subject of the transaction, so close-#1 is the closing of account number 1. Some events have a third part too, denoting an amount associated with the transaction. Thus payin-#1-$50 is a deposit of $50 to account number 1. The second and third components of event names are attributes that qualify the transaction name: they are not to be thought of as arguments of functions or procedures. Also there is no expression of causality; although the customer probably chooses how much to deposit, and the system generates the balance report, the traces do not make this distinction. The trace

<close-#1-$100>

is not a trace of the banking system because an account cannot be closed if it has not been opened, and

<open-#1-$100, payin-#1-$50, bal-#1-$200>

cannot occur because the balance reported is not correct.

We now look at two ways of specifying the banking system formally.

[1] We chose the model-oriented style of VDM for the data specification rather than the two-tiered approach of Larch to make the example as small and self-contained as possible. The CSP specifications omit process alphabets because of the complication of channel events.

2.2.2 The Data Specification

We start by specifying the *state* of the system, and we choose a mapping from account identifiers to account balances, initially empty:

State is ACCS: AccID $\mapsto$ Dollars **initially** $\{\}$

We assume that the types *AccID* and *Dollars* are defined, and that *Dollars* includes only non-negative sums (so overdrafts are never allowed). The map is a set of pairs, so that, if, for example, we represented both *AccID* and *Dollars* as natural numbers,

$$\text{ACCS} = \{\ 1 \mapsto 50,\ 5 \mapsto 18\}$$

would denote the state of the system in which there are two accounts with account identifiers 1 and 5, and balances of \$50 and \$18 respectively. The balance of account 1 would then be given by

$$\text{ACCS}\ (1) = 50$$

The domain of the mapping would be

$$\text{dom}\ (\text{ACCS}) = \{1,\ 5\}$$

The balance of an account whose identifier is not in the domain is not defined, so

$$\text{ACCS}\ (3)$$

has no meaning in this case.

Now we define the *operations* one by one. The open operation takes an account identifier and an initial deposit as *arguments*, and adds a new pair to the mapping. The operation is not defined if an account with that identifier already exists. The $\oplus$ operator adds a new pair to the mapping, overriding any existing pair with the same first element.

open (n: AccID, d: Dollars)
  requires $n \notin \text{dom}\ (\text{ACCS})$
  ensures $\text{ACCS}' = \text{ACCS} \oplus \{n \mapsto d\}$

The close operation takes only the account identifier, and removes the corresponding account from the mapping. If an account with that identifier does not already exist, the effect of the operation's execution is not defined.

close (n: Num)
  requires $n \in \text{dom}\ (\text{ACCS})$
  ensures $\text{ACCS}' = \text{ACCS} \ominus n$

The payin operation increments the balance related to a given account identifier, by replacing the pair mapping the account identifier to the old balance with a pair mapping it to the new balance.

payin (n: AccID, d: Dollars)
requires n $\in$ dom (ACCS)
ensures ACCS′ = ACCS $\oplus$ {n $\mapsto$ (ACCS (n) + d)}

The wdraw operation decrements the balance, and is undefined when the account does not exist or when the amount withdrawn exceeds the balance.

wdraw (n: AccID, d: Dollars)
requires n $\in$ dom (ACCS) & d $\leq$ ACCS (n)
ensures ACCS′ = ACCS $\oplus$ {n $\mapsto$ (ACCS (n) − d)}

The bal operation returns the balance and leaves the mapping unchanged

bal (n: Num) d: Dollars
requires n $\in$ dom (ACCS)
ensures d = ACCS (n) & ACCS′ = ACCS

This is the only operation that has an *output* – the syntax of the operation's signature distinguishes the input variables (inside the brackets) from the output variables (outside the brackets). For convenience, we shall call both input and output variables *arguments.*

Commentary

1. **State** The state of the banking system is central to the specification. By presenting it first, we were able to define each operation in terms of its effect on the state. This gives modularity; each operation can be understood alone, without reference to other operations.

   In this trivial example, the state has only a single component. Generally, the state has many components, and the operations are organized according to which component they access. These sets of operations are called *types.* It is also common for an operation to be implemented with more primitive operations, so a system may be structured as a hierarchy of types. For example, the types *Dollars* and *AccID* would be specified as types used by the specification above.

   Note that the state is *abstract.* We have not said how it is represented. We could implement the mapping ACCS as a list or a hash table, for example, so long as the implementation type satisfies the properties of the abstract type. The types *AccID* and *Dollar* have not been defined. However, we are assuming that we can add and subtract *Dollars*, and that we can test equality of values of *AccID.*

2. **Sequencing** The specification does not constrain the order in which the operations may occur. Any operation may occur, with any arguments, at any time. This does not seem to coincide with our trace specification above, which asserts that an account cannot be closed before it is opened. However, the **requires** clauses do

restrict the order of executions in which the effect of each operation is *defined.* It is common to assume that the operations will only be used when they are defined, and so there is at least an implicit sequencing [2].

Nevertheless, even if we make this assumption, the sequencing constraints are obscure. We can only tell that an account must be opened before it is closed by analyzing the **requires** and **effect** clauses of *all* the operations – and yet this is an important feature of the system. Sequencing constraints relate operations, and are not well expressed as local properties of operations.

3. **The Nature of an Event** The data paradigm views an event as an execution of an operation with zero or more arguments. An operation is *invoked* by one object on another, so that the object providing the operation cannot influence when it is executed. The arguments are divided into inputs and outputs; input values are chosen externally, and output values are chosen internally. The notion of control flow is rigid, and it matches that of most imperative programming languages.

2.2.3 The Process Specification

The first step in building a process specification is to identify the most fundamental event ordering in the system. Here, it is the ordering of events in a single account. So we specify an account process, ACC, by the equations[3]

$$\begin{aligned} \mathrm{ACC} &= \mathrm{open?}\ d \rightarrow \mathrm{ACC}_d \\ \mathrm{ACC}_b &= \mathrm{payin?}\ p \rightarrow \mathrm{ACC}_{b+p} \\ &\quad \mid \mathrm{wdraw?}\ w \rightarrow \mathrm{ACC}_{b-w} \\ &\quad \mid \mathrm{bal!}\ b \rightarrow \mathrm{ACC}_b \\ &\quad \mid \mathrm{close} \rightarrow \mathrm{STOP} \end{aligned}$$

$$(d > 0, p > 0, 0 < w \leq b)$$

The words in upper case are names of processes, and those in lower case are names of events, channels and variables. The arrow is the 'prefix' operator: the process expression

$$\mathrm{e} \rightarrow \mathrm{P}$$

denotes the process that first engages in the event e and then behaves like the process P. STOP is the process that cannot engage in any event. The vertical bar indicates choice; after an open event, the process may engage in a payin, a wdraw, a bal or a close.

The question marks denote inputs, and the exclamation marks outputs. The mark is preceded by a channel name, and followed by the message. So bal!100 denotes the output

[2] Another interpretation is that an error must be signalled when an operation is invoked with a false **requires** clause (see, for example, [Cohen 86]).

[3] Strictly, we should specify the alphabets of our processes. We have chosen not to, since in all our examples the alphabet does not include events not in the process equations. Also, the presence of communication events makes denoting the alphabet tedious.

of 100 on channel bal. The channel events are no different from the other events. CSP views the channel name and the transmitted value as two parts of the name of a single event. A process selects an output value for a channel by offering only one event for that channel; on input the process offers a choice of events, one for each value it is willing to receive on the channel. So here, payin? p denotes a whole set of events, one event for each value of p. (The range of values the variables may take is given beneath the process equations.)

The equations may be thought of as a schema for a much larger set of equations in which no free variables (such as p) occur: there would be one instance of the second equation for each value of the balance b, and each branch of the choice prefixed by an input would be split into many branches, one for each possible value of the variable associated with the channel name.

We can deduce from ACC's specification the set of traces that it may engage in. The set includes, for example,

<open.10, close>
<open.100, wdraw.75, payin.20>
<open.20, payin.100, bal.120, close>

The banking system consists of many such accounts, so we label the process ACC with the account identifier. Let us assume that the account identifiers are just integers. Then the process for the account whose number is 253 is written

253:ACC

and it behaves just like ACC, but when ACC would have engaged in some event e, 253:ACC engages in 253.e. So the traces of 253:ACC, corresponding to the traces of ACC above, are

<253.open.10, 253.close>
<253.open.100, 253.wdraw.75, 253.payin.20>
<253.open.20, 253.payin.100, 253.bal.120, 253.close>

The system is the parallel combination of the account processes. If there are n accounts, we would write

BANK = $\|_{i \leq n}$ i:ACC

The parallel combinator enforces synchronization on shared events, but otherwise allows the processes to run independently. Since none of the account processes share any events, there is no synchronization, and the executions of the processes are interleaved arbitrarily.

Commentary

1. **State** There is no mention of state in the specification. The process equations can be given an operational semantics in which the process names denote states, and

$e \rightarrow P$ denotes a state transition on event e[4]. But Hoare prefers to avoid this, and is careful to read $e \rightarrow P$ as 'e then P', the process that engages in the event e and then behaves like P. This is significant, for state is not a central concept in the process paradigm. As a result, expressing the balance of an account is tricky. The indexing of process names is an *ad hoc* technique that Hoare does not define rigorously in his book on CSP [Hoare 85]. His examples of indexing are restricted to simple types (such as integers and sets) and it is not clear how the technique can be generalized. Furthermore, considering the index to be part of the process *name* has unintuitive consequences. Two account processes with the same account number but different balances are, in the formalism, no more similar than accounts with different account numbers. There is a similar problem in the treatment of events discussed below.

2. **Sequencing** Instead of emphasising state, like the data specification, the process specification emphasizes the sequencing of events. It is quite clear from the specification, for example, that an **open** must be the first event of an account. However, the indexing of process names does obscure the sequencing a little. As in the data specification, to establish that a sequence in which a withdrawal leaves the balance negative is illegal, we would have to analyze the values of the indices. We can think of the sequencing in two parts: a gross ordering of events ('open, transactions, close') and further constraints to do with the balance part of the state. The merit of the process specification is that it separates these constraints: we can recover the gross ordering by deleting the indices.

   The process specification uses explicit concurrency to combine the partial orderings on the events of each account. The data specification, on the other hand, was silent on the subject of concurrency.

3. **The Nature of an Event** The event names are not decomposed into operations and arguments. The account number and transaction amount are just parts of the name; they only qualify the kind of event in a syntactic sense. There is also no semantic distinction between input events and output events. The '?' and '!' suffices are not part of the event name, but are syntactic sugar for more complex process expressions. The expression $c?x \rightarrow P(x)$, *eg*, is short for the choice

   $$c.x_1 \rightarrow P(x_1) \mid c.x_2 \rightarrow P(x_2) \mid \ldots c.x_n \rightarrow P(x_n)$$

   where $x_1, x_2, \ldots, x_n$ are the possible values of x.

   A process is either willing or unwilling to engage in an event. There is no sense in which one process chooses one part of the event name, and a different process chooses another. Despite its elegance, this notion of communication is often quite unintuitive. The process formalism is powerful enough to describe more elaborate forms of communication (using the deterministic and non-deterministic choice

[4]More accurately, each process *expression* denotes a state. The transition relation is built by induction on the syntactic structure of the process expression, using a deductive system in the sense of logic. This semantics is due to Plotkin, and is well explained in [Olderog 86].

operators explicitly), but communication is such a central issue that it should be treated explicitly in the semantics.

The view of an event as something that processes participate in, with no notion of the event being caused or chosen by one process and suffered by another, has some advantages. We do not yet know who initiates a balance enquiry: perhaps the account process initiates the event itself following a deposit or withdrawal. The operation structure of the data specification is thus an implementation bias that the process specification avoids.

2.2.4 Elaborating The Specifications

We have discussed briefly some of the differences between the two specifications, and criticized some of their features. These features may seem insignificant, but their repercussions are enormous in larger specifications. In this section, we shall look at some reasonable elaborations of the banking system, and show how each elaboration is much better accommodated by one of the two specifications. The evidence we give for the relative merits of the paradigms is scant. We hope, though, that the reader will have seen similar cases in real projects, and that these examples will provide a new way of viewing problems that are already familiar.

State

Suppose the bank charges a fee of \$1 for every withdrawal of over \$1000. We can amend the data specification by elaborating the post-condition of the wdraw operation[5]:

wdraw (n: AccID, d: Dollars)
requires $n \in dom\ (ACCS)$ **and** $newbal \geq 0$
ensures $ACCS' = ACCS \ominus n \cup \{n \mapsto newbal\}$
where $newbal = ACCS\ (n) - d - fee$
$fee =$ **if** $d > 1000$ **then** 1 **else** 0

Modifying the process specification would be harder. It is not clear that the indexing mechanism could bear the description of the occasional fee; perhaps we should separate the wdraw events into those that incur a fee and those that do not (which would be untidy). The concept of an operation works well in this case: it allows us to divide the occurrences of wdraw cleanly into two sets that modify the state differently. This example demonstrates that the notion of state may well be fundamental to the system, and that a system built on the process paradigm may suffer from the omission of an explicit state.

The data specification gives the *global* state of the system. Sometimes this is handy. Some functions are much easier to express in terms of a global state. Suppose the bank manager wants to audit the total funds of the bank. We could add an operation audit that sums the total of the balances of all the accounts[6]:

[5]The scope of the 'where' clause extends over both **requires** and **ensures**

[6]An omitted requires clause is interpreted as 'true'.

audit () tot: Dollars
ensures tot = $\Sigma_{i \in dom(ACCS)}$ ACCS (i)

The process specification would require a new process. Each process might need a new event to report the balance to the auditing process (if, for example, the bank charged a fee for balance enquiries by customers). We might want to say that the balances of the individual accounts could be obtained in any order, and although this could be expressed using non-deterministic choice, it would be cumbersome.

The data paradigm allows us to state invariants on the global state. For example, we could assert that the ratio of deposits to loans for the whole bank exceeds a minimum legal requirement, without specifying how that ratio is maintained. In the process paradigm we could add a process to maintain the ratio, but it would suffer from the same awkwardness as the auditing process above.

The notion of explicit global state has disadvantages too. There is no reason that different accounts should not be accessed concurrently, and yet, if the global state (ACCS) is encapsulated in the implementation (as the specification seems to suggest it should be), accessing different accounts at once will lead to contention. This problem arises in the 'data modelling' approach to development, which prescribes the design of a data model (often as a database schema) as the first step. The developer must then consider 'consistency constraints' that ensure that the data model is kept in a meaningful state. This may be a self-inflicted problem; complex constraints are often the result of simple process orderings.

Sequencing

Perhaps the banking system always reports an account balance after a deposit or a withdrawal. This change in the event sequence is simple for the process specification; we just insert some **bal** events:

$$
\begin{aligned}
&\mathrm{ACC} = \mathrm{open?a} \rightarrow \mathrm{ACC}_a \\
&\mathrm{ACC}_b = \mathrm{payin?}\ p \rightarrow \mathrm{bal!(b+p)} \rightarrow \mathrm{ACC}_{b+p} \\
&\qquad\qquad \mid \mathrm{wdraw?}\ w \rightarrow \mathrm{bal!(b-w)} \rightarrow \mathrm{ACC}_{b-w} \\
&\qquad\qquad \mid \mathrm{bal!}\ b \rightarrow \mathrm{ACC}_b \\
&\qquad\qquad \mid \mathrm{close} \rightarrow \mathrm{STOP}
\end{aligned}
$$

$$(d > 0, p > 0, 0 < w \leq b)$$

The data specification's implicit notion of sequencing breaks down here. Even if we added a flag to the state that made every operation except **bal** undefined after a **wdraw** or a **payin**, the specification would be stilted. The control structure imposed by operations is also an obstacle here; it is no longer clear who performs the **bal** operation.

Abstraction of Sequencing

Suppose the banking system is to be modified to allow more than one account per customer. Each customer has two accounts, a checking account and a savings account. The checking account is opened with an initial deposit of \$100 or more. Whenever its balance drops below \$100, \$500 is transferred automatically from the companion savings account. Assuming a balance enquiry follows each transaction (as above), we could label the savings account of a customer with s and the checking account with c, and add the process[7]

$$\text{CUST} = \text{c.bal?a} \rightarrow \text{s.wdraw!500} \rightarrow \text{c.payin!500} \rightarrow \text{CUST} \quad (\text{a} < 100)$$

Each customer has a customer number, i say, and accounts labelled $i.s$ (savings) and $i.c$ (checking). The new system is given by

$$\text{BANK} = \|_{i \leq n}\ i{:}(\text{CUST} \parallel \text{s:ACC} \parallel \text{c:ACC})$$

This change is hard for the data specification. Although the state could be extended to describe customers, this sort of interaction is messy to specify. There is no operation that could accommodate this feature. Suppose, for example, that we chose to modify the withdraw operation of the checking account, so that when a withdrawal is made that leaves the balance below \$100, an operation is invoked for the transfer. This would only postpone the problem until the bank manager decides that balance enquiries should incur a fee (and may thus move the balance below the limit). The data specification is ill-suited to properties that do not belong to single operations.

We can interpret this problem as a more general criticism of the data approach: a lack of abstraction of scheduling constraints. There is no explicit concurrency, and data objects may only interact by invoking each other's operations. This usually confines a system developed in the data paradigm to a hierarchical structure, which is frequently over-restrictive. In particular, there is no way to express cooperation – every activity must be executed by some object on another.[8]

This has one of two consequences in the design of large systems. In order not to overspecify the scheduling, a designer may leave the modules of a system unconnected (perhaps just specifying which 'uses' which) even at a late stage in design, when their interaction should be understood. Alternatively, the scheduling may come hand-in-hand with the module design (as in the 'top-down' approach). Low-level (and premature) scheduling concerns then dominate the earliest design decisions, and the specification structure is twisted to fit the control structures of the implementation. Jackson has argued, moreover, that top-down design is methodologically unsound [Jackson 76]. The availability of information, and the risk and cost of error, should determine the order in which design decisions are made. A mistake in choosing the scheduling of the largest components of the system is expensive, and yet the top-down approach advocates making

[7]Note that if there are insufficient funds in the savings account, the transfer cannot be made, and the system would deadlock. Of course, this would be undesirable in a real system.

[8]The message-passing paradigm of object-oriented approaches is more powerful, but suffers from lack of abstraction: we do not want to embed scheduling decisions within the definitions of methods.

that decision when the information it depends on (namely, what is being scheduled) is not available.

Abstraction of State

There is a corresponding lack of abstraction in the process paradigm: we cannot define abstract states. Although the notion of state can be avoided completely, we have seen that it is useful – for example, in describing the balance of an account. Even if we succeed (by indexing of process names, for instance) in incorporating a notion of state, it is likely to be crude. The account processes are indexed by integers; the data specification allowed us to omit the detail of how account identifiers are represented. Suppose for example, we wanted to distinguish domestic accounts from foreign accounts by their account identifiers. We could specify an operation that decided whether an account was foreign or domestic without specifying how account identifiers are represented. The process paradigm would force us to make a premature decision, choosing perhaps to assign odd numbers to foreign accounts and even numbers to domestic accounts.

In large systems, this has two bad effects. It biases the specification, introducing into an abstract design detailed algorithms for manipulating data structures. Secondly, it does not enforce the encapsulation of data types; the result is code that is unmaintainable because users of data types may rely on quirks of their representation.

2.2.5 Conclusions

We examined some features of the two paradigms, and showed that each paradigm is only well-suited to describing some aspects of a system. We have argued that the deficiencies of the paradigms may lead to unmaintainable systems. These deficiencies are not just linguistic, but are often due to the way the paradigms are used. For example, while it is clear that the separation of operations and their arguments is basic to the data paradigm, the fact that the notion of an operation leads to a hierarchical structure (and thus perhaps over-restrictive scheduling) can only be deduced from an understanding of how a data specification is commonly used and interpreted.

The data paradigm is based on the notion of *state*, while the process paradigm is based on *sequencing*. Both are useful notions that are likely to arise in any system. Without explicit state, a system cannot take advantage of representation independence, a vital tool for reducing complexity, and without sequencing notions, the modules of a system cannot be connected in an abstract fashion.

2.3 Our Approach

We advocate using both paradigms together. In the next chapter, we present a model in which the data and process aspects of a system are specified as separate objects. Our primitive event is a pair of an operation and an argument, but objects interact by a CSP-like parallel combinator. Data objects are denoted in a syntax that makes state explicit; the process syntax uses regular expressions to describe sequencing constraints.

Properties of operations may be encapsulated within the operation definitions of data objects. This allows us to specify a system in which sequencing and state are both given substantial treatment, so that we can benefit from the best features of each paradigm.

Chapter 3

The Model

In this chapter, we present a simple example of a specification in two parts, data and process. We give a semantics that translates each part into an object; the objects are then combined according to simple set-theoretic rules. The section on building systems shows, by several small examples, how the technique may be extended to larger collections of objects.

There is no semantic distinction between process objects and data objects. A single semantic domain provides a common framework in which to combine parts of a specification written in two distinct notations. The combination of two objects is itself an object. The whole system is thus thought of as an object, but we will not want to classify it as a data or a process object. Indeed, our contention that both paradigms are needed rests on the assumption that it will be hard to describe a system as a single data or process object.

We do not expect our specifications to be efficiently executable, and thus we do not advocate our specification language for implementations. However, we assume that any implementation language can be given a semantics in terms of our object model. Therefore, following our discussion of the model, we define a satisfaction ordering on objects, which provides the foundation for future work on implementation.

3.1 A Ticket Machine

We start by considering an informal specification of a simple ticket machine to motivate a more formal specification that follows. We give an informal, operational interpretation to the formal specification, and then present a formal model in terms of traces.

3.1.1 Informal Description

As part of an effort to make travel more convenient, the London Underground provides ticket-dispensing machines at most stations. Here is an informal account, typically incomplete and ambiguous, of the behaviour of a ticket machine:

> The machine serves customers one by one. The customer selects a fare; the 'insert coins' light flashes; the customer inserts one or more coins; the ticket

is delivered; and, sometimes, one or more coins are returned in change.

Two fares are available, 60p and 80p. The machine accepts 10p, 20p, 50p and £1 coins. The ticket is not delivered until at least the required fare has been inserted. If more is inserted, change is returned in an arbitrary sequence of 10p and 20p coins.

The machine engages in a sequence of events, for example, selection of an 80p fare, flashing the insert light, insertion of a £1 coin, delivery of an 80p ticket, return of 20p in change. Each event has two components, an operation, for example 'deliver a ticket', and an argument, '80p'.

Notice that the description is in two parts. Each constrains the ordering of events in some way. The first part gives a gross ordering: a cycle of a selection, a flash of the light, coin insertions, a ticket delivery and return of change. The second part adds a constraint for the events of each operation – for example, that ticket delivery cannot occur until the fare has been inserted, and that the fares available are 60p and 80p.

3.1.2 Formal Specification

This leads us to a formal specification in two parts, which denote a *process object* and a *data object*, shown in Figure 3.1. Later, we shall show that the ticket machine can be thought of as a single object, composed of these two component objects. Start by looking at the specification of the process object, TM_p. The operations are given with their argument types (that is, all the possible values of their arguments). This tells us what the possible events are. The light operation has only one event, and so only a single argument. The timetable is a regular expression[1] denoting a language of operation sentences. Any sequence that is a prefix (proper of improper) of a sequence in the language is a possible operation sequence of TM_p. Some examples are

'select, light'
'select, light, incoin, deliver'
'select, light, incoin, incoin, deliver, select'

The specification of the data object, TM_d, resembles a model-oriented specification of an abstract data type. First a mutable state is defined: here, it has two components, the balance in favour of the customer (BAL), and the selected fare (FARE). Then the operations are defined one by one. Each has a *signature* giving its argument type. The *context* clause denotes the set of states in which the operation may occur. Note that it differs from the *requires* clause of the data specification in Chapter 2. First, it is a firing condition and not a 'precondition': the operation cannot occur if it is false. Secondly, the condition depends only on the state, and not the argument of the operation. The *effect* clause relates the state after execution of the operation (primed) to the state before and to the argument. So, for example, incoin can occur when the balance is less than the fare,

[1] Regular expressions are explained in [Hopcroft 86]. e^* means a sequence of zero or more e's; e^+ means a sequence of one or more e's, and $e \square f$ (used later) means e or f. Appendix C contains a glossary of symbols and terms. The use of regular expressions here is not significant; many other formalisms could be used.

$TM_p \triangleq$ **Operations are**
select?: {60,80}
light: {flash}
incoin?: {10,20,50,100}
deliver?: {60,80}
outcoin?: {10,20}

Timetable is (select? light incoin?$^+$ deliver? outcoin?*)*

$TM_d \triangleq$ **State is**
BAL: {10,20,30,...,150} **initially** 0
FARE: {60,80} **initially** 60

Operations are

select? f: {60,80}
context BAL = 0
effect FARE′ = f

incoin? c: {10,20,50,100}
context BAL < FARE
effect BAL′ = BAL + c

deliver! f: {60,80}
context BAL ≥ FARE
effect BAL′ = BAL − FARE **and** f = FARE

outcoin! c: {10,20}
context BAL > 0
effect BAL′ = BAL − c **and** c ≤ BAL

Figure 3.1: A formal specification of the ticket machine

and its effect is to increase the balance by the amount inserted. When state components are omitted from the context, they may take any value (so select can occur whatever the value of FARE), and when they are omitted from the effect, they are unaltered (so select does not alter the value of BAL). Note that more than one post-state/output argument pair may satisfy the effect clause of an output operation: the effect of outcoin allows coins to be returned in any sequence of 10p and 20p pieces that reduces the balance to zero.

The operations of an object are partitioned into *input* operations, marked '?', *output* operations, marked '!', and *internal* operations, which are unmarked[2]. Internal operations are unique to an object, and occur under its control. The process object has an internal operation light that it performs autonomously following select. The input and output operations comprise the *external* operations, and occur only with the participation of other objects. Argument values are selected by output operations. An operation may be an input of one object and an output of another (eg. deliver) or an input of both objects (eg. incoin), but may not be an output of more than one object. For this reason, deliver is an *input* of the process object – it is the data object that selects the argument value.

The meaning of the combined specification is obtained as follows. The history of the ticket machine up to some moment is given by the sequence of events, or *trace*, that it has engaged in. Its overall behaviour is described by the set of all possible traces. Each object is interpreted as a set of traces, and the traces of the combination are all those satisfying both objects. A trace satisfies an object if the subtrace consisting of events whose operations belong to the object is a trace of the object. In Section 3.3 we present this formally in a trace semantics that abstracts from the concerns of how such a specification would be executed. However, to motivate the more abstract approach, we start by discussing our operational model.

3.2 The Operational Model

Recall that the operations of an object are divided into external operations, namely the inputs and outputs, and internal operations. Internal operations are executed by an object autonomously, and without the participation of other objects. External operations may be shared, and their execution is synchronized between objects.

3.2.1 Communication

The execution of operations is governed by the same rule as the CSP parallel combinator. Suppose we have two objects, P and Q. Operations that they share require simultaneous participation of both P and Q. However, operations belonging to P but not to Q are of no concern to Q, which cannot control them or even notice them. Such operations may occur independently of Q whenever P engages in them. Similarly, operations of Q not belonging to P may occur without P's participation.

[2]The notation is taken from CSP, but is used differently. In our model, the marks are used simply to identify input and output operations. In CSP, they are used as a syntactic sugar in the process syntax, and a channel (operation) may be used for input at one point in the process execution, and output at another.

A system consisting of several objects may itself be regarded as an object. We can think of an object in a system as interacting with an *environment* object representing the rest of the system. At some moment, an object offers a set of external operations, or *options*. The environment offers an option set too. Any external operation that is in both option sets may occur. (The execution of internal operations of an object is independent of the environment, and is discussed in 3.2.2.) When an external operation occurs for an object, it is either an input or an output. If it is an output, the object selects an argument value, and delivers that value to the environment. If it is an input, it receives a value from the environment [3] Whatever the value of the argument, the object cannot refuse it. This is fundamental to our model. Operations are chosen by negotiation between objects, but arguments are selected by one object alone.

Here is an example. Suppose the ticket machine has just engaged in an incoin operation. According to the process object, either incoin or deliver may happen next: the option set is {incoin, deliver}. The option set of the data object may be {incoin} or {deliver}, depending on whether the money that has been inserted exceeds the fare due. In the former case, {incoin} is the intersection of the option sets, and thus the ticket machine is prepared only to receive further coins from its customer. In the latter case, the intersection is {deliver}, and it may deliver the ticket. The deliver operation is an output of TM_d and an input of TM_p, and so TM_d selects the argument value.

The combination of the two component objects of the ticket machine is itself an object. The interaction between the ticket machine and the customer follows the same pattern. If the customer is prepared to do an incoin, and the machine is willing too, then the operation can occur. It is an output for the customer and an input for the ticket machine, so the customer chooses the argument (that is, a coin), and passes it to the ticket machine (by placing the coin in the slot).

3.2.2 Internal Operations

In simple cases, the operations an object can perform next are all either external operations or internal operations. If they are all external, the environment can influence the choice of next operation; if they are all internal, the object selects one autonomously and performs it. Sometimes, there will be both internal and external operations that can occur. In that case, which happens next is not determined. (We can think of it depending on the relative timing of the object and its environment. If the environment presents one of the external operations 'before' the object has performed an internal operation, then the external operation will occur).

A comparison with CSP may be helpful. The internal operations correspond to hidden events in CSP. The process

$$P = (\sqcap_i\, x_i \rightarrow P_i') \,\Box\, (\,\Box_j\, y_j \rightarrow P_j'')\backslash\{x_i\}$$

offers a set of external events y_j on its first step. However, it can perform any of the hidden events x_i, and whether one of the external events occurs depends on when the

[3] If there is no corresponding output operation (which would not be the case in practice), the value is chosen non-deterministically.

event was offered by the environment. If a hidden event occurred first, an external event y_k can only then occur if it is an external event of P_i'. The difference between our model and CSP is that our internal operations appear in the traces, but hidden events in CSP do not.

3.2.3 Crash Events

The reader may be disturbed that, since all the arguments of an allowed input operation must be acceptable, the implementer of a data object has no freedom in dealing with exceptional argument values. However, this is not the case: the effects relation may be partial. When an operation occurs with an argument for which no next state exists satisfying the effects relation, the object *crashes*. Crashing is described by a special internal operation ($\star$). No events can follow a crash, so if an object crashes, it brings the whole system down.

Later, when we view the object as a specification (in Section 3.5), we shall allow an implementation to behave as it pleases when the specification crashes. Of course, if we do not know how an implementation is going to behave, it might as well crash. The notion of crashing is useful when we compare two specifications: one that behaves sensibly when the other crashes is preferable. The inclusion of crashing in the model does not imply that we want it to be present in the systems we specify. To the contrary, we need the notion of crashing in our model so that we can show, within our formalism, that it does *not* happen!

3.2.4 Separation of Operation & Argument

There are two reasons that we favour the distinction between operations and arguments. We think of communication as the passing of information in some *direction*, but we want to retain the expressive power of a communication that can be refused. The separation of operation from argument permits both goals to be achieved. Secondly, we are suspicious of primitives (as in CSP) that allow a receiver to decline a communication on account of its argument. The argument is the *content* of the communication; it seems to make no sense to talk of a communication being refused after it has already happened.

One might argue that excluding arguments from contexts limits the expressive power of the model. This is partly true. Suppose we were specifying a bank account, with a withdraw operation that takes as an argument the amount requested. We cannot say that the withdraw event does not occur if the amount requested is greater than the account balance. Instead, we would use two operations, withdraw-req? and withdraw-grant!. Whether or not the second occurs depends on the state of the account alone, which has been extended to include the amount requested by the previous operation.[4]

[4]Alternatively, we could provide an operation name for each amount requested, eg. withdraw10. CSP deals with arguments in this way. We contend that the separation of operation from argument is a vital semantic issue, not captured by a syntactic sugaring.

3.3 The Trace Model

We now give a trace-based semantics for objects. So far, we have presented a syntax for data objects, and a syntax for process objects, and have mentioned composing a process object and a data object. There are semantic objects which have no representation as data or process objects; a system is thought of as a single semantic object denoted by a set of combined syntactic specifications.

First we define semantic objects and their combination. Then we give the meaning function informally, explaining how the semantic objects corresponding to the syntactic data and process objects are obtained. Because of some intricacies that arise for the data object, we treat its semantics formally. Finally, we define a satisfaction ordering on semantic objects, the first step towards implementation.

The semantic model is an extension of the basic (ie., no hiding, refusals or divergences) CSP trace model . It differs from CSP in three ways. First, the separation of operation and argument is embedded semantically in our model, but is treated as a syntactic issue in CSP. Secondly, our model has internal events, which, like the hidden events of CSP, are selected unilaterally by an object and executed autonomously, but, unlike hidden events, appear in traces. Thirdly, we have a notion of crash events: if a crash occurs in an object, the whole system is disabled. Combined with our definition of satisfaction (which allows an implementation to behave as it pleases instead of crashing), this gives us a crude, but adequate, expression of chaotic behaviour without the complication of refusals and divergences. Non-determinism in the choice of operation is limited to internal operations, but we can simulate non-determinism in external operations.

The reader should be aware that we are not proposing this model as a competing theory of concurrency and communication. It is a simple, workable model that can be used to give meaning to our specifications.

3.3.1 Definition of an Object

A semantic object is a 5-tuple

$$(OP, A, OUT, INT, T)$$

where the components are:

- A set of operation names OP.
- An argument mapping A. $A(op)$ is the set of argument values associated with the operation op.
- A set of output operations $OUT \subseteq OP$.
- A set of internal operations $INT \subseteq OP$, disjoint from the output operations.
- A trace set T. A trace is a finite sequence of events; an event is a pair of an operation and an argument.

We define the *external* operations to be those that are not internal, and the *input* operations to be the external operations that are not outputs

$$EXT = OP - INT = IN \cup OUT$$

Operations With Only One Event

Sometimes an operation has only one event. Then its argument type contains only one value, and since the value is immaterial, we write the single event as just the operation name. In the specification of the ticket machine (Figure 3.1) we gave the light operation the argument flash to avoid confusion. By this convention, we would write both the operation and its event as light. (Also, in the specifications that follow, we shall omit the argument types of operations with only a single event.)

Internal Operations

An object may have an internal operation signifying a catastrophe that leaves the object disabled. This operation has a single event, *crash*, written $\star$. Since objects never share internal operations, we sometimes want to indicate that it is the crash of object X, and we then write $\star_x$.

Trace Set Properties

The trace set satisfies five properties:

1. **Contains Empty Trace** Any object can do nothing, so the empty trace is a member of T.

   $$<> \in T$$

2. **Prefix Closed** A prefix of a history must also be a history

   $$s \cdot t \in T \;\Rightarrow\; s \in T$$

   where $s \cdot t$ is the concatenation of traces s and t.

3. **Events Well-Typed** The set of events the object can engage in, its *legal events*, are given by

   $$E(OP, A) \mathrel{\widehat{=}} \{op.a \;:\; op \in OP \;\wedge\; a \in A(op)\}$$

   Every trace must be a sequence of legal events. Denoting by E^* the set of all finite sequences whose elements are members of the set E,

   $$T \subseteq E(OP, A)^*$$

4. **Input Arguments Enabled** If an input operation can occur, then it can occur with any argument of the right type

   $$\forall t, op, a \;.\; (op \in IN \;\wedge\; t<op.a> \in T) \;\Rightarrow\; (\{op\} \times A(op)) \;\subseteq\; T//t$$

   where $T//t$ is the set of events that may follow t in T [5]

[5] Let $<e>$ denote the trace consisting of the single event e. Then $T//t = \{e \;:\; t<e> \in T\}$.

5. **Crash Fatal** Nothing can follow a crash event

$$\forall t : T \;.\; T//t<\star> = \emptyset$$

Thus, if a crash event occurs in a trace, it must be the last event. It is useful to define the *legal traces* of operation set *OP* and argument mapping *A*, *LT(OP,A)*, to be the set of traces whose events are well-typed and in which crash is fatal.

3.3.2 Combining Objects

Compatible Objects

Suppose we have some set of objects $\{X_i\}$. We shall denote the component C of X_i by C_i, so, for example, OP_k is the operation set of the k-th object.

The objects are *compatible* when

1. Internal operations are not shared

$$INT_i \cap OP_j = \emptyset, \text{ if } i \neq j$$

2. No operation is an output of two objects

$$OUT_i \cap OUT_j = \emptyset, \text{ if } i \neq j$$

3. If *op* is an output of X_i and an input of X_j, the argument type of *op* in X_i is a subset of the argument type of *op* in X_j

$$\forall op : OUT_i \cap IN_j \;.\; A_i(op) \subseteq A_j(op)$$

An object is not compatible with itself. If it were, two instances of the same object could choose different output arguments or internal operations. For simplicity, our model requires that only one object select the output argument of an operation, and that internal operations may be executed autonomously.

Semantics of Combination

We now give a combination rule for objects. Intuitively, the combination is a parallel composition similar to that of CSP. However, instead of synchronizing on shared events, objects synchronize on shared operations. Compatibility guarantees that any argument selected by the output operation of one object will be acceptable to the corresponding input operation of the other object.

If the objects $\{X_i\}$ are not compatible, then their combination is not defined. If they are compatible, the combination is given by

$$(OP, A, OUT, INT, T)$$

where

- An operation of an object is an operation of the combination:

$$OP \triangleq \cup_i OP_i$$

- The argument type of each operation is the intersection of the types in the component objects:

$$A(op) \triangleq \cap_{i\,:\,op \in OP_i} A_i(op)$$

- If an operation is an output for one object, it is an output of the combination:

$$OUT \triangleq \cup_i OUT_i$$

- If an operation is internal to an object, it is internal to the combination:

$$INT \triangleq \cup_i INT_i$$

- A trace is a trace of the combination if it is a legal trace, and if the subtrace that each object X_i participates in, $t\,|_{OP_i}$, is a trace of that object

$$T \triangleq \{t \in LT(OP, A) \;:\; \forall i \;.\; t\,|_{OP_i} \in T_i\}$$

  $t\,|_S$ is the subtrace of t restricted to the events whose operations are in the set S.

Note (from the definition of the legal traces) that if a crash event is in a trace, it must be the last event. So, if one object crashes, the combination rule dictates that the combination crashes.

Denoting Combinations: Meshing and Linking

When two objects are combined, they are synchronized on common operations. If they share some operations, we say that they are *linked* by those operations; if they share none, we say they are *meshed.* It is useful to write the combinator in two different ways. When X and Y are linked, the combination is written

$$X \odot Y$$

and when they are meshed, the combination is written

$$X \oslash Y$$

The combinator (however it is written) is associative and commutative.

3.4 The Trace Semantics

We now consider the trace semantics of the ticket machine specification of Figure 3.1. Each part of the specification denotes an object. The partitions of the operation set are obtained by examining the markings of the operations: the outputs are all those marked '!', and the internal operations are unmarked. The declarations of the operations and their argument types give the legal events the object can engage in.

3.4.1 Traces of the Process Object

The traces of the process object are all the legal traces whose operations are in the order given by the timetable (including the empty trace), e.g.,

<>
<select.60>
<select.60, light, incoin.10, deliver.80>
<select.60, light, incoin.20, deliver.60, outcoin.20>
<select.80, light, incoin.100, deliver.60, select.80>

The process object is ignorant of argument values. Although the gross ordering of operations make sense, the process object allows more change to be returned than was inserted, and does not necessarily deliver a ticket of the fare requested.

3.4.2 Traces of the Data Object

The traces of the data object are obtained inductively. The first operation may be any whose context is true of the initial state. If that operation is an output, then its argument may be any that satisfies the effect clause (for some post state). If the operation is an input, then any argument of the type is allowed. So we obtain a set of events that can occur on the first step, and for each event, a new state[6]. We thus build the trace set by a breadth-first search of the event tree. These are some traces of TM_d

<>
<select.60>
<select.60, select.80>
<select.80, deliver.80>
<select.60, incoin.100, outcoin.20>

Unlike the traces of the process object, these are financially sound, and the fare of a delivered ticket is always that of the last request. However, the data object does not respect the gross ordering of operations, *eg*, change may be returned before a ticket is delivered.

[6]There must be exactly one such state. Section 3.4.5 explains this in more detail.

3.4.3 The Traces of the Combination

The combination of TM_p and TM_d is a linking, because they share operations. The ticket machine composed of the two objects is written

$$TM = TM_p \odot TM_d$$

Here are some of its traces:

<>
<select.60>
<select.60, light, incoin.50>
<select.60, light, incoin.20, incoin.50, deliver.60, outcoin.10>
<select.60, light, incoin.10, incoin.50, deliver.60, select>

Consider the fourth trace. We can easily check that it is a trace of the combination. The events are well-typed, crash is fatal (since there is no crash event in it), and so it is a legal trace. All its operations are operations of the process object, so the process object must engage in every event. So we check that it is a trace of the process object, which it is, since the order of operations is consistent with the timetable. The data object does not have the light operation, so we remove it, and check that the remaining subtrace

<select.60, incoin.20, incoin.50, deliver.60, outcoin.10>

is a trace of the data object, which indeed it is.

3.4.4 Crashes

There is a complication in generating the trace set of a data object. We stipulated that having chosen an input operation we can give it any well-typed argument, and calculate the new state of the object. There may be arguments for which no next state exists (satisfying the effect clause). We call these *bad* arguments, and the input is a bad input. Having appended a bad input to a trace, we include the trace obtained by appending the crash event $\star$, and extend it no further. The set of internal operations is also augmented with the crash operation.

Consider TM_d. It has been designed so that when an input operation occurs any argument value is acceptable. But suppose we had omitted the context clause of incoin (an omitted context being taken as true for all states), writing instead:

incoin? c: {10,20,50,100}
effect BAL′ = BAL + c

Now suppose BAL = 100, and incoin.100 occurs. There is no value of BAL′ that can satisfy the effect clause. The trace set would then allow only $\star$ as the next event, and no subsequent events. So, a trace of TM_d would be

<select.60, incoin.100, incoin.100, $\star$>

and after this trace the object is dead, unable to engage in any event. This is bad: a system should never crash. The designer of the system must ensure that crashes are not possible – the model provides the notion of crashing precisely so that one can show that it never happens. However, it may be possible to design a component of the system so that it is much more efficient if it may do as it pleases on some 'bad' inputs. The designer would then specify that the component crashes when those inputs occur, but would take care to ensure that they never could. The satisfaction ordering we define later (in Section 3.5) allows an implementation to do anything when the specification says it may crash. The implementer of the component is thus given the freedom to treat the bad inputs in the most convenient way.

3.4.5 The Semantics of Data Specifications

In this section we formalize the derivation of semantic objects from syntactic data specifications[7]. We are not concerned with concrete syntax, and we assume that we have already extracted some auxiliary semantic entities.

We start with a set of operations $OP = \{op_i\}$ partitioned into inputs IN, outputs OUT and internal operations INT, a set of states Σ, and an initial state $\sigma_0 \in \Sigma$. Each operation op_i has a set of arguments A_i, a context that is a predicate on states

$$context_i : \Sigma \rightarrow Bool$$

and an effect that is a predicate on (pre-state, argument, post-state) triples

$$effect_i : \Sigma \times A_i \times \Sigma \rightarrow Bool$$

Restrictions

We impose two requirements on data specifications[8]. When an output operation can occur, that is, the context is true of some pre-state, there must be at least one output argument/post-state pair satisfying the effect:

$$\forall op_i : OUT, \sigma : \Sigma \;.\; context_i(\sigma) \Rightarrow \exists a : A_i,\; \sigma' : \Sigma \;.\; effect_i(\sigma, a, \sigma')$$

We cannot allow there to be two post-states satisfying the effect for the same pre-state/argument pair, for otherwise, the object may refuse an operation when, according to the trace set, it is a legitimate extension to the trace[9]:

$$\forall op_i : OP, \sigma : \Sigma \;.\; context_i(\sigma) \Rightarrow$$

$$\neg\exists a : A_i,\; \sigma', \sigma'' : \Sigma \;.\; effect_i(\sigma, a, \sigma') \wedge effect_i(\sigma, a, \sigma'') \wedge \sigma' \neq \sigma''$$

[7]For process specifications, the informal description should suffice.

[8]Both of these requirements are over-restrictive, since there may be pre-states of an operation that could never occur. Modularity, however, dictates that we judge the operations independently of one another.

[9]This constraint is very strong; it could be discarded if our model handled full non-determinism. See Section 5.3.1.

Deriving Traces

The second restriction guarantees that each trace has at most one associated state. This allows us to define a function Q from traces to states, with a special element *broken* appended:

$$Q : T \rightarrow \Sigma \cup \{broken\}$$

Informally, $Q(t)$ is the state of the data object after it has engaged in the trace t. The traces of the data object, T, and the function Q are derived by a simultaneous induction:

1. The empty trace <>is a trace of the data object; its state is the initial state

$$Q(<>) = \sigma_0$$

2. If t is a trace whose state satisfies the context of some operation op_i, and (a, σ) is an argument/post-state pair satisfying the effect of op_i, then we can obtain a new trace by appending $op_i.a$ to t

$$\frac{t \in T,\ context_i(Q(t)),\ a \in A_i,\ effect_i(Q(t), a, \sigma)}{t \cdot <op_i.a> \in T,\ Q(t \cdot <op_i.a>) = \sigma}$$

3. If t is a trace whose state satisfies the context of some input operation op_i, and a is an argument for which there is no post state satisfying the effect then we can append $op_i.a$ to t, but the data object is broken thereafter

$$\frac{t \in T,\ context_i(Q(t)),\ op_i \in IN,\ a \in A_i,\ \neg\exists \sigma : \Sigma\ .\ effect_i(Q(t), a, \sigma)}{t \cdot <op_i.a> \in T,\ Q(t \cdot <op_i.a>) = broken}$$

4. When the data object is broken, it crashes

$$\frac{t \in T,\ Q(t) = broken}{t \cdot <\star> \in T}$$

The traces are exactly the traces that can be derived with these rules.

3.5 Satisfaction

In this section we examine what it means for an implementation to *satisfy* a specification. Demonstrating that an implementation obeys the constraints of a specification has two parts: translating the behaviour and structure[10] of the implementation into the language of the specification, and verifying that the constraints are indeed obeyed. For the ticket machine, the former must necessarily be informal: insertion of a coin, for example, maps to incoin. But for a program, a formal construction is in order. To show that an array, for example, faithfully models an abstract set, we would give an *abstraction function* that maps each array to a set. When the programming paradigm does not match that of the specification, providing such a mapping can be hard. The implementer of a CSP specification who chooses to program in Pascal rather than in Occam faces a host of extra problems. We shall not, therefore, discuss this mapping here– it would amount to giving a semantics of a programming language in our object model.

3.5.1 The Satisfaction Ordering

We shall address the question of satisfaction, therefore, in its most fundamental form: when does an object X satisfy an object Y?

Clearly an object satisfies itself, so the satisfaction relation is reflexive. Stepwise development requires it to be transitive, and the distinction between a specification and an implementation suggests it should be anti-symmetric too. Satisfaction is thus a partial ordering, and when X satisfies Y, we write

$$X \sqsubseteq Y$$

3.5.2 Satisfaction Properties

Basic Properties

The most basic property a specification demands of an implementation is *safety*: whatever the implementation does, the specification could do. For example, an implementation of the ticket machine may not deliver a ticket if payment has not been made: the trace

<deliver.60>

is not a trace of TM. However, in one case we allow the behaviour of an implementation to differ from that of its specification. If the specification object crashes, the implementation object may do *anything*. So our first criterion for satisfaction, $X \sqsubseteq Y$, is that any trace in T_x that does not have a proper prefix r that can be followed by a crash in T_y must be a trace of T_y. Thus

$$\{t \in T_x \ : \ \forall r{<}t \ . \star \notin T_y/r\} \subseteq T_y$$

[10]We discuss in Section 5.1.1 why a specification should impose structural as well as behavioural constraints.

where T/t is the set of operations that may follow trace t in trace set T[11]. The crash event here is any object's crash event; we can think of $\star$ as a wildcard matching any crash event. We shall refer to this condition as the *trace subset law.*

A more subtle safety property is that an object must not have too few external operations. Suppose an object has an operation *op*, but it appears in no trace. Then, according to the specification, the object will prevent that operation from occurring in a combination. So *op* must be an operation of the implementation too. Conversely, an operation not present in the operation set of the specification must be omitted from the implementation too, for otherwise the implementation will block that operation, and may thus fail to allow some legal behaviour. This is our first *liveness* property. Safety properties dictate what an implementation must *not* do; liveness properties dictate what they *must* do. The combination of the safety and liveness properties for operations thus require that the sets match

$$OP_x = OP_y$$

A safe implementation must not make an input operation an output; a live implementation must not make an output an input. So outputs in the specification must be outputs in the implementation

$$OUT_x = OUT_y$$

Similarly, the internal operations of X must be the internal operations of Y

$$INT_x = INT_y$$

Remember that our internal operations are visible: they occur in the object's traces. Because they are visible, we do not even allow renaming. A ticket machine that followed incoin with ringbell instead of light would not satisfy the specification. Internal operations introduce the notion of autonomy, not hiding, into our model.

If the specification is compatible with some object, the implementation must be too. This leads us to the constraint on A_x. Input argument types may be widened, and output argument types may be narrowed: so the argument mapping of the implementation must satisfy

$$\forall op : IN \;.\; A_x(op) \supseteq A_y(op)$$

$$\forall op : OUT \;.\; A_x(op) \subseteq A_y(op)$$

Since the traces of X must be legal sequences, input operations must accept all well-typed arguments. So the first constraint on A_x affects T_x: whenever an input operation can occur in X, it accepts at least the arguments it would have accepted in Y.

Trace Liveness Properties

The trace subset law ensures that the traces of the implementation are safe. They must also be live. If the specification offers a choice of external operations after some trace, then, if that trace is a trace of the implementation, the implementation must offer at least as much choice

$$\forall t \in T_x \;.\; T_x/t \cap EXT \supseteq T_y/t \cap EXT$$

[11] $T/t = \{op \;:\; \exists a \;.\; t{<}op.a{>} \in T\}$. Recall that $T//t$ is the set of *events* following t in T.

The internal operations of an object are selected autonomously by the object. Since the environment cannot influence the choice, it is non-deterministic, and thus the implementation may offer only a subset of the internal operations. However, if the specification offers some operations, the implementation must offer at least one of them

$$\forall t \in T_x \,.\, T_y/t \neq \emptyset \;\wedge\; \star \notin T_y/t \;\;\Rightarrow\;\; T_x/t \neq \emptyset$$

Summary of Satisfaction Properties

There are thus two freedoms an implementation can exploit. It may reduce the internal choices of output argument and internal operations. When the specification crashes, the implementation may stop, or continue to work, behaving in any way.

3.5.3 Compositionality of Satisfaction

It is very important that the notion of satisfaction be context independent; an implementation of an object must be valid irrespective of the environment in which it is placed. Then we can build the system object by object, knowing that if each individual implementation object satisfies its specification, then their combination satisfies the specification of the system.

A satisfaction ordering with this property is said to be *compositional.* Formally, the ordering $\sqsubseteq$ is compositional if

$$X \sqsubseteq Y \;\;\Rightarrow\;\; C[X] \sqsubseteq C[Y]$$

for any context C and any objects X and Y. Since we have only introduced one composition operator (and since combination is commutative and satisfaction is transitive), it suffices to show that for all X, Y and Z

$$X \sqsubseteq Y \;\;\Rightarrow\;\; X.Z \sqsubseteq Y.Z$$

where '.' denotes $\oslash$ or $\odot$ (depending on the choice of X, Y and Z).

Consider first the cases in which the objects are deterministic (no internal operations, and at most one output argument for a given trace and appended operation). Then satisfaction requires the objects to be identical in everything but their argument mappings, and since subset and superset are preserved by set intersection, satisfaction is preserved. Internal operations are not a complication, since each internal operation can belong to at most one object. The only subtlety is that reducing the choice of output argument might introduce deadlock. By the definition of an object, when an input operation can occur it can occur with all its arguments. Since satisfaction requires each input operation of an implementation to have at least the argument set of the specification, reducing the choice of output arguments for some trace and appended operation cannot introduce deadlock.

3.5.4 Examples

Reducing Output Argument Choice

An implementation of the ticket machine TM may order the coins it returns in change as it pleases, since the choice of output arguments is internal. So an implementation of TM_d, TM'_d say, may define outcoin as

outcoin! c: {10,20}
context BAL > 0
effect BAL′ = BAL − c **and** c ≤ BAL **and** BAL ≥ 20 ⇒ c = 20

and would satisfy the specification: $TM'_d \sqsubseteq TM_d$. In a sequence of change, this implementation always delivers 20p coins first, and then, if necessary, a 10p coin.

Reducing Operation Choice

Internal operations are chosen non-deterministically. The non-determinism may be resolved when the system is built (by the implementer) or when it runs (by a hidden mechanism). Suppose we had specified the ticket machine process as

$TM_p' \;\widehat{=}$ **Operations are**
select?: {60,80}
green, red
incoin?: {10,20,50,100}
deliver?: {60,80}
outcoin?: {10,20}

Timetable is (select? (green incoin?$^+$ deliver? outcoin?*) ▯ red)*

We have replaced the internal operation light with two internal operations, green and red (we have omitted their argument types since each operation has only one event). The new machine can decide arbitrarily to refuse to supply tickets. When the user has selected the fare, either the green light comes on, indicating that the machine will supply the ticket, or the red light comes in, in which case the only operation the machine will allow next is another fare selection.

Non-determinism in specifications never requires the implementation to behave unpredictably. It is usually a choice offered to the *implementer*, who will take advantage of the freedom, often to increase efficiency. This specification allows the machine to run out of tickets (and be refilled); the mechanism by which it is emptied and replenished is not modelled.

One reasonable implementation would be to ensure that the machine is refilled before it runs out, so that the machine will always supply tickets:

$TM_p'' \;\widehat{=}$ **Operations are**
select?: {60,80}

green, red
incoin?: {10,20,50,100}
deliver?: {60,80}
outcoin?: {10,20}

Timetable is (select? green incoin?$^+$ deliver? outcoin?*)*

Because choice of internal operation may be reduced, $TM''_p \sqsubseteq TM'_p$. Another legal (but unreasonable) implementation would be to flash the red light every time, never selling any tickets.

Surviving Instead of Crashing

A data object for calculating harmonic means provides three operations: clear? signals the start of a new sample, enter? records a value, and mean! delivers the mean of the last sample.

MEAN $\widehat{=}$ **State is**
T: Real **initially** 0
N: Nat **initially** 0

Operations are
clear?
effect $N' = 0$ **and** $T' = 0$

enter? e: Nat
effect $T' = T + 1/e$ **and** $N' = N + 1$

mean! m: Real
context $N \neq 0$
effect $m = N/T$

The state component N keeps a count of the sample size, and T is the cumulative sum of the reciprocals of the values entered in the latest sample. The mean cannot be requested unless the sample contains at least one value.

Consider what happens if enter.0 occurs. Then there is no value of T' satisfying the effect condition, since 1/0 is undefined. The trace set of MEAN will thus contain, for example,

<clear, enter.0, $\star$>

Note that we cannot prevent enter.0 happening by amending the context of enter. Once an argument has been passed, the operation has occurred. On the other hand, the context of mean 'protects' its effect clause by preventing the operation occurring when its state, *before* execution of the operation, is inappropriate.

A better specification would prescribe that, instead of crashing, the object give a mean of zero if zero is entered in the sample:

MEAN′ ≙ **State is**
T: Real **initially** 0
N: Nat **initially** 0
Z: Bool **initially** false

Operations are
clear?
effect N′ = 0 **and** T′ = 0 **and** not Z′

enter? e: Nat
effect N′ = N + 1 and
if e = 0 **then** Z′ **and** T′ = T
else T′ = T + 1/e **and** Z′ = Z

mean! m: Real
context N ≠ 0
effect if Z **then** m = 0 **else** m = N/T

The state component Z records whether a value of zero was entered in the last sample. The trace set of MEAN′ then contains, for example,

<clear, enter.0, mean.0>

and because an implementation may behave in any way after a trace that ended in crash for the specification, MEAN′ ⊑ MEAN.

3.6 Building Systems

Now that we have presented our semantic model, we can demonstrate its use in some more elaborate examples. We start by showing how a process object can be combined with more than one data object, and then consider combinations with more than one process. The use of data objects as 'channels' is illustrated. We also introduce two new operators that are useful in building systems: enclosure and renaming.

3.6.1 More Than One Data Object

Suppose we wanted to add an internal operation refresh to the ticket machine that updates a display (not specified) showing the total number of tickets sold of each fare. Then we could specify a data object that keeps separate counts of the number of 60p tickets sold (C60) and the number of 80p tickets sold (C80)[12]:

CT $\hat{=}$ **State is**
C60: Nat **initially** 0
C80: Nat **initially** 0

Operations are
refresh. c: Nat-pair
effect c = (C60,C80)

deliver? f: {60,80}
effect
f = 60 $\Rightarrow$ C60′ = C60 + 1
f = 80 $\Rightarrow$ C80′ = C80 + 1

The new system is given by the combination

$$\mathrm{TM}_p \odot \mathrm{TM}_d \odot \mathrm{CT}$$

Note that TM[13] and CT are linked only by the deliver operation, and that CT does not constrain its occurrence. CT does not affect the orderings of the operations in TM. The refresh operation is performed autonomously by CT at unspecified times.

Fairness

This specification does not guarantee that the display of tickets sold is up to date all the time. If refresh never occurs, the display is never updated. However, it does ensure that it is up-to-date just after the display has been refreshed (by execution of a refresh operation). How often refresh occurs is not specified. Our model does not provide any way of saying it occurs 'often enough'. Introducing such notions of *fairness* into the

[12]Recall that state components omitted from the effect are unaltered.
[13]Recall that $\mathrm{TM} = \mathrm{TM}_p \odot \mathrm{TM}_d$.

model would complicate the treatment of non-determinism, compromising the simplicity of our model. We believe that, for the sort of systems we are specifying, the standard elaboration of the trace model for fairness, namely considering infinite as well as finite traces, would be inadequate. The station manager does not want to know that the ticket machine will *eventually* update its display. Sometimes the fairness property will be definable only in terms of notions outside the model: for example, that the display is updated as soon as electrical connection permits. Otherwise it will be more appropriate to formulate it as a straightforward safety property. For example, we might provide a clock process with an event end-of-day, and require that a refresh event occur at least once between each pair of end-of-day events.

3.6.2 Making Output Operations Internal

Suppose we wanted to specify that the display is updated after every ticket delivery. Then we might amend the process object to offer refresh after deliver

$\mathrm{TM}_p \;\widehat{=}\;$ **Operations are**
select?: {60,80}
light
incoin?: {10,20,50,100}
deliver?: {60,80}
outcoin?: {10,20}
refresh

Timetable is (select? light incoin?$^+$ deliver? refresh outcoin?*)*

This object is not compatible with CT, though: they share the internal operation refresh. We want refresh to be an external operation of the individual objects, but an internal operation of their combination (since refresh is executed autonomously by the machine as a whole).

So we make refresh an input operation of TM_p, and an output of CT. Then we *enclose* the operation refresh, rendering it internal. The new ticket machine is denoted

$$\Diamond_{\{refresh\}}\ (\mathrm{TM}_p \odot \mathrm{TM}_d \odot \mathrm{CT})$$

Definition of Enclosure

Given a semantic object X, its enclosure around a set of output operations $C \subseteq OUT_x$ is given by

$$\Diamond_C X \;\widehat{=}\; (OP_x, A_x, OUT_x - C, INT_x \cup C, T_x)$$

The operations that are converted must be outputs, for otherwise there will be internal operations for which no object selects the argument.

3.6.3 More Than One Process Object

Perhaps the ticket machine contains a clock, which we represent as a process

CLK $\triangleq$ **Operations are** tick! tock!

Timetable is (tick! tock!)*

We can use the clock to time the updates of the display, by adding another process that is willing to do refresh once in every five clock periods

DISP $\triangleq$ **Operations are**
tick?
refresh?: Nat-pair

Timetable is (tick? tick? tick? tick? tick? refresh?)*

Combining the clock, the display process, the counter and the basic machine of Figure 3.1, we obtain

$$\text{OTM} = ((\text{CLK} \odot \text{DISP}) \oslash (\text{TM}_p \odot \text{TM}_d)) \odot \text{CT}$$

Recall that we write $\odot$ when objects share operations, and $\oslash$ when they do not. Arranging the combination like this highlights the interactions or *linkings*, between objects. The use of $\oslash$ shows that the basic ticket machine TM is not linked directly to CLK or DISP. They are loosely synchronized by the counter CT, which mediates between them. The fewer links there are between objects, the easier the combination is to understand – we therefore arrange the expression so that as many of the operators are $\oslash$ as possible.

The system is depicted in Figure 3.2. Data objects are drawn as circles, and process objects as boxes. Linking operations are represented by lines between the linked objects; an arrow indicates that the operation is an output for the object at its tail. We can always tell from the diagram whether two objects interact directly – we just see if there is a line joining them. More care needs to be taken when we write combinations as expressions. The expression

$$\text{OTM} = ((\text{CLK} \odot \text{DISP}) \odot (\text{CT} \odot \text{TM}_d)) \odot \text{TM}_p$$

denotes the same system as the one above, but we cannot infer from it that neither TM_p nor TM_d are directly linked to CLK or DISP.

The operations refresh, tick and tock are internal to the system, and so we enclose them obtaining

$$\text{CTM} = \diamond_{\{refresh, tick, tock\}} \text{OTM}$$

3.6.4 Data Objects As Channels

We can think of the counter as a channel along which information passes. The deliver operation is like a write, and the refresh is like a read.

This paradigm is suitable for describing many kinds of asynchronous communication. The two communicating objects, X and Y say, usually do not share any operations, so we mesh them and link the combination to a data object, D say, obtaining

$$(X \oslash Y) \odot D$$

For example, FQ is a FIFO queue data object

FQ $\hat{=}$ **State is** Q: Int-Queue **initially** newq

Operations are
enq? i: Int
effect Q′ = append(Q,i)

deq! i: Int
context ¬ empty(Q)
effect Q′ = rest(Q) **and** i = first(Q)

W is a data object that enqueues consecutive powers of 2

W $\hat{=}$ **State is** P: Int **initially** 1
Operations are
enq! p: Int
effect p = P **and** P′ = 2 * P

R is a process that dequeues integers

R $\hat{=}$ **Operations are** deq?: Int
Timetable is deq?*

The combination (W $\oslash$ R) $\odot$ FQ denotes a system in which consecutive powers of 2 are enqueued and dequeued. The values are dequeued in the order they were enqueued in, and no value is dequeued before it is enqueued. Some traces of the combination are

<>
<enq.1, enq.2, deq.1>
<enq.1, deq.1, enq.2, enq.4, enq.8, deq.2>
<enq.1, enq.2, enq.4, enq.8, enq.16, enq.32>

Synchronizing Channels

The degree of asynchrony the channel permits depends on the specification of the channel data object. If the data object is a FIFO queue, we get a standard asynchronous stream communication (as above). A bounded queue would synchronize the processes more tightly; a unary buffer would force reads and writes to alternate.

Using Process Objects In Channels

The unary buffer example raises an interesting question: where should we state the property that reads and writes must alternate? It can be placed in the data object, or in a separate process object. The data object would impose alternation implicitly in the context conditions. A state component would be declared with values **full** and **empty**; the context of the read would require the flag to be set to **full**, and the context of the write would require it to be **empty**; and the effect clauses of the write and read would assert values of **full** and **empty** after execution. The second approach would restrict the operation scheduling in a process timetable instead of using the flag: we would link the buffer data object to a process with timetable

(write read)*

The arrangement would be denoted

(WRITER $\oslash$ READER) $\odot$ (ALT $\odot$ BUF)

where BUF is the buffer data object and ALT is the process object forcing alternation of reads and writes. See Figure 3.3.

It is not clear which approach is preferable here. We hope that, in general, scheduling constraints will be inherent in the state of the data object or quite unrelated to it, and it will then be clear where to place them. For example, the constraint that an empty stack may not be popped should be in the context of the **pop** operation; the constraint that a bank account should be opened before it is closed should be described in a timetable. Some examples, like this one, do not fall immediately into one category or the other, and the specifier may choose either.

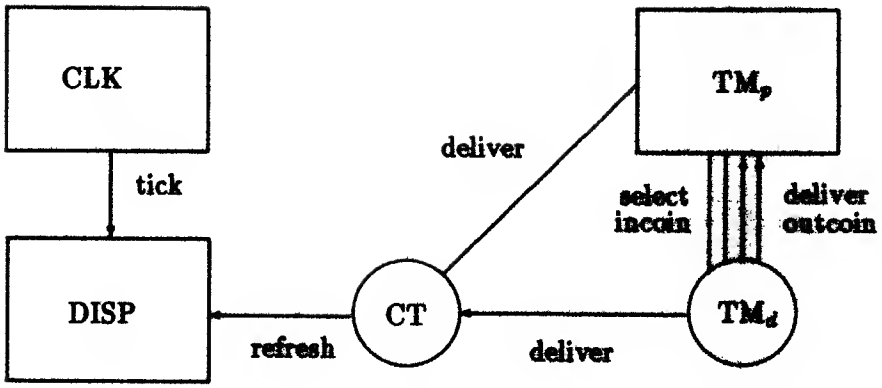

Figure 3.2: Ticket machine with display updated by clock

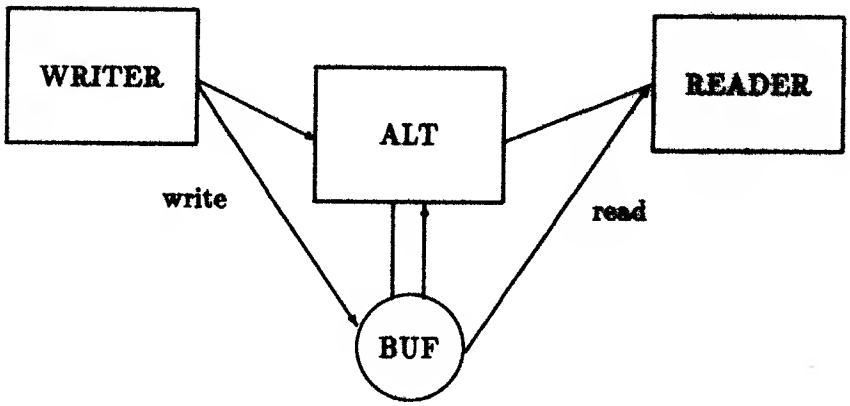

Figure 3.3: Using a process to enforce alternation of reads and writes

3.6.5 Renaming

Labelling

Most stations provide more than one ticket machine. A pair of ticket machines standing side by side is given by the meshing

$$2TM = left{:}TM \oslash right{:}TM$$

The ticket machines have been *labelled.* Labelling TM with 'left' prefixes its operation names and the events in its traces with 'left:'; wherever TM would have engaged in *op.a*, left:TM may engage in *left:op.a.* So a trace of 2TM is

<right:select.80, right:light, left:select.60, right:incoin.20, left:light>

Because the labels differ, the two machines have disjoint operation sets and there is no synchronization between them.

Uses of Renaming

Labelling is a simple case of renaming. We allow a more general form of renaming in which each operation is replaced by one or more 'pseudonyms'. There are four principal uses of renaming:

1. *Describing a group of similar objects by instantiating a generic object.* Labelling is generally used for this, as in the pair of ticket machines above. The account objects of a banking system, for example, might be labelled with account numbers.

2. *Re-use of a general purpose object.* Many objects recur in software designs, and it may be convenient to re-use an object that has been specified before, changing its operation names to suit the new context. For example, we might use a FIFO queue as a message stream, giving the operation names **enq** and **deq** new names **writemsg** and **readmsg**.

3. *Giving different local names to shared operations.* We have seen how several objects may share operations. Often, extra clarity is gained by giving the shared operations different names in different objects. Consider, for example, linking a customer process to the ticket machine. In the customer process, the operation synchronizing with the **deliver** operation of the ticket machine might be called **taketicket**. To synchronize the two operations, we could rename **taketicket** to **deliver**, or vice-versa, or rename both operations to **deliver-and-take**, say.

4. *Addressing an operation with more than one name.* A process object may want to use more than one name for a single operation of a data object it is linked to.

   An automatic teller machine (ATM), for example, modelled as a process object, may provide two operations for balance enquiries, **dispbal** for screen display and **printbal** for printed acknowledgment. An account data object may just provide an operation **balance** that we want to link to both operations. The account data object would be renamed, giving **balance** the pseudonyms **dispbal** and **printbal**.

Formal Definition of Renaming

Let f be a surjection[14] that maps a set of operation names U to the set of operation names of object X

$$f : U \rightarrow OP_x$$

We define $Rename(f, X)$, the renaming of X with f, so that whenever X may engage in an operation op, $Rename(f, X)$ may engage in any operation u such that $f(u) = op$. These operations are *pseudonyms* of op under f. It follows that

$$\begin{aligned} Rename(f, X) = (U, A_x \circ f, \quad & \{u \in U \ : \ f(u) \in OUT_x\}, \\ & \{u \in U \ : \ f(u) \in INT_x\}, \\ & \{t \in LT(U, A_x \circ f) \ : \ f^*(t) \in T_x) \end{aligned}$$

where the extension of f to traces, f^* is defined by

$$\begin{aligned} f^*(<>) &= <> \\ f^*(<u.a>) &= <f(u).a> \\ f^*(st) &= f^*(s)f^*(t) \end{aligned}$$

Note that f maps new names to old names.

The renaming must retain the meaning of the crash operation, so we require that, if $\star \in OP_x$ then $\star$ has only one pseudonym, $\star'$ say,

$$f(u) = \star_x \ \Rightarrow \ u = \star'$$

We can now define labelling. The labelled object l:X is the same as $f_l(X)$, where the domain of the renaming f_l is the set whose elements are those of OP_x prefixed with 'l:', the range is OP_x, and

$$f_l(l : op) = op$$

3.6.6 Examples

Two examples of renaming follow. Each uses labelling to describe a group of similar objects (purpose 1 above). In addition, the first demonstrates the addressing of an operation with more than one name (purpose 4), and the second demonstrates synchronizing operations with distinct names (purpose 3).

Bank

A specification of a bank with one ATM and n customers, each with a single account, is shown in Figure 3.4.

The account data objects are labelled with account numbers ranging over $1 \ldots n$. The label of the incard operation identifies the account number (which is recorded on the card). The operation balance has two pseudonyms, dispbal and printbal.

[14] A surjection is a function that is onto its range: for each value in the range, there is a domain value that maps to it.

ATM $\widehat{=}$ **Operations are**
reqcash?: Dollars
delivercash?: Dollars
deposit?: Dollars
dispbal?: Dollars
printbal?: Dollars
incard?, outcard!

Timetable is
($\Box_{i \leq n}$ (i:incard?
(i:deposit?
$\Box$ i:printbal?
$\Box$ i:dispbal?
$\Box$ (i:reqcash? i:delivercash?))
outcard!))*

ACC $\widehat{=}$ **State is**
BAL: Dollars **initially** 0
C: Dollars **initially** 0

Operations are
balance! b: Dollars
effect b = BAL

deposit? d: Dollars
effect BAL′ = BAL + d

reqcash? r: Dollars
effect C′ = min(r,BAL)

delivercash! c: Dollars
effect c = C **and** C′ = 0 **and** BAL′ = BAL − c

BANK = ATM $\odot$ ($\oslash_{i \leq n}$ i:ACC [dispbal, printbal for balance])

Figure 3.4: A banking system with one ATM and n accounts

[14] A surjection is a function that is onto its range: for each value in the range, there is a domain value that maps to it.

Instead of writing the renaming of dispbal and printbal as f(ACC) and defining f separately, we have written

ACC [dispbal, printbal for balance]

which is equivalent to f(ACC), where f maps dispbal and printbal to balance, and every other operation name of ACC to itself. Another notational convenience is the distributed use of $\oslash$ and $\Box$.

The number of accounts n is a constant; we cannot create objects dynamically. This is not a problem in practice — n is just a bound on the number of account objects in the system, and accounts exist before they are opened.

Queue Merger

We take the FIFO queue, FQ, given before, and define three instances by renaming: 1:FQ, 2:FQ and m:FQ. A merging process M dequeues values from 1:FQ and 2:FQ

M $\hat{=}$ **Operations are**
1:deq? : Int
2:deq? : Int

Timetable is (1:deq $\Box$ 2:deq)*

By renaming m:enq to 1:deq and 2:deq, values are enqueued on m:FQ as they are dequeued from 1:FQ and 2:FQ. The queue merger is given by

$$MQ = \Diamond_{\{1:deq,2:deq\}} (M \odot (1:FQ \oslash 2:FQ) \odot FQM)$$

where

FQM = m:FQ [1:deq, 2:deq for m:enq]

Having enclosed 1:deq and 2:deq, the remaining external operations are 1:enq, 2:enq and m:deq. MQ is a non-deterministic merge. The order of values enqueued on one of the queues is the order in which they will be dequeued, but the interleaving of the two streams is non-deterministic – there is no guarantee that a value enqueued with 1:enq before a value is enqueued with 2:enq will be dequeued first. Not even fairness is guaranteed: a value enqueued with 1:enq may never be dequeued. A sample trace of MQ is[15]

<1:enq.3, 1:deq.3, 2:enq.5, 1:enq.7, 1:deq.7, 2:deq.5, m:deq.3, m:deq.7>

M is redundant, since its timetable does not restrict the ordering of operations at all. Its inclusion is justified on the grounds of maintainability. Suppose instead we wanted a deterministic merge that took elements from the two queues alternately, starting with 1:FQ. The object combination would be the same, but we would change the timetable of M to

Timetable is (1:deq 2:deq)*

[15]It would be convenient to hide the internal operations here, but we have avoided hiding because it introduces more non-determinism than our model can handle. See Section 5.3.1.

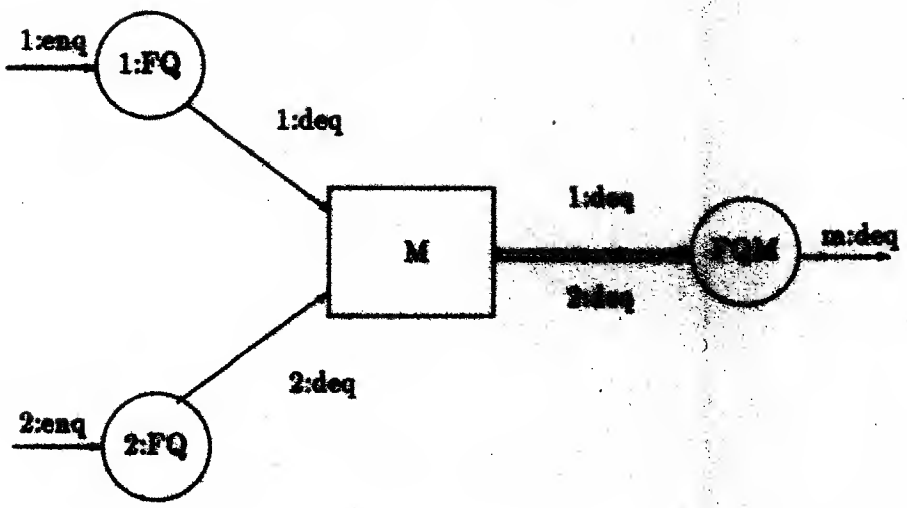

Figure 3.5: Non-deterministic merge

Chapter 4

A Case Study

In this chapter, we specify and implement a small program. The example serves three purposes. First, it sheds light on the meaning of satisfaction, in particular by showing how operations can be implemented using procedures. Secondly, it may reinforce the reader's understanding of our (informal) operational semantics. Thirdly, it is a starting point for a wider discussion of implementation concerns.

The specification is an elaboration of the ticket machine of Chapter 3. Two new features are added to the ticket machine: more realistic handling of change, and the reporting of ticket sales. We also take the opportunity to present a slightly more elaborate data specification. Previously, we assumed that the types and operators that we used (for example, integers with plus and minus, queues with dequeue, enqueue and size) are understood. If the state has a more complex type (say, binary tree or priority queue), this is no longer adequate. We show how this can be remedied by pre-defining the types and their operators in an accompanying algebraic specification.

The implementation is entirely sequential – although we simulate concurrency – and is written in CLU. It gives a flavour of the (reasonably conventional) style of specification we have in mind. However, the style of implementation is not well developed, and much work remains to be done before we could advocate it as a practical technique.

4.1 The Specification

The specification we shall implement is shown in Figure 4.1. There are three objects: a ticket machine process TM_p, a ticket machine data object TM_d, and an auditing process AUDIT. The processes do not share operations, and both are connected to the data object:

$$\mathrm{TM} = (\mathrm{TM}_p \oslash \mathrm{AUDIT}) \odot \mathrm{TM}_d$$

The ticket machine objects are almost identical to the first example of Chapter 3 (Figure 3.1). We have added two simple features:

1. Instead of an unbounded supply of change, the machine can only return coins that have previously been inserted. An out-of-change light warns the customers if there might not be enough change for the next transaction.

2. The machine maintains a count of the number of tickets sold of each fare. The auditing process may examine the counts and reset them periodically.

4.1.1 The Specification of TM_d

A new state component (COINS) records the number of coins available of each denomination, which is modified by the incoin and outcoin operations. As before, we have not specified exactly how change is to be given: we just require that the coins returned have been previously inserted. The context of outcoin guarantees that it is only executed when a suitable coin is available.

The state macros are just a syntactic mechanism to improve the clarity of the specification. They may contain no free variables aside from the state variables.

The out-of-change light is not specified explicitly. We have added an operation, setlight, that switches the light on and off. It outputs a boolean value, *true* if there is not enough change, and *false* otherwise. The operation's context is the negation of outcoin's context: when change is due, and the machine can deliver some of it, it may not perform setlight. The effect of setlight is non-deterministic. When the machine cannot guarantee that there will be enough change for the next transaction ($\neg$ *guarantee-change*), it *must* output 'true', switching the out-of-change light on. When there is an abundance of change (*lots-of-change*), it must output 'false', switching the light off. The implementer is thus free to choose the point at which the change level is deemed too low, so long as it falls between these bounds.

We decided, arbitrarily, that there is 'lots of change' when there are 100 10p pieces in the machine. The machine can guarantee delivering enough change for any transaction if, for any possible change amount, there is a selection from the coin supply whose total equals that amount.

The state component TIX maintains the number of tickets sold of each fare. The new operation report outputs the counts of tickets sold of each fare. The operation can only be executed when at least 6 tickets have been sold since the last report, and its effect is to reset the counts to zero. (There are many more realistic ways to time the execution of report, but they would add nothing interesting here.)

The type assignments and the coercion operator nat are explained in Section 4.1.4.

4.1.2 The Specification of TM_p

The process specification differs from the example of Chapter 3 only in the addition of the new operation setlight. Note that, because of its placement, the context condition of the setlight data operation acts as a termination condition for the iteration of outcoin[1]. Thus, when change is due, but there is no suitable coin in the machine to deliver, execution of outcoin may be omitted. If the out-of-change light was off at the start of the transaction, it is guaranteed that whatever change is due will be delivered.

[1]The termination condition is deterministic in this example because the contexts of outcoin and
have previously been inserted. An out-of-change light warns the customers if there might not be enough change for the next transaction.

$TM_p \triangleq$ **Operations are**
setlight?: Bool
select?, deliver?: Fare
incoin?, outcoin?: Coin
Timetable is (setlight? select? incoin?$^+$ deliver? outcoin?*)*

AUDIT $\triangleq$ **Operations are** report?: Fare-Mset
Timetable is report?*

$TM_d \triangleq$ **State is**
BAL: Nat **initially** 0
FARE: Fare **initially** 60
COINS: Coin-Mset **initially** new
TIX: Fare-Mset **initially** new

State Macros are
outcoin-poss = $\exists$ c: Coin . count (COINS, c) > 0 & nat (c) $\leq$ BAL
guarantee-change =
$\forall$ n:Nat . $\exists$ cs: Coin-Mset . possChangeAmt (n) $\Rightarrow$ (total (cs) = n & (cs $\subseteq$ COINS))
lots-of-change = count (COINS, 10) > 100

Operations are
setlight! b: Bool
context $\neg$ outcoin-poss
effect ($\neg$ guarantee-change $\Rightarrow$ b) & (lots-of-change $\Rightarrow$ $\neg$ b)

report! r: Fare-Mset
context size (TIX) $\geq$ 6
effect r = TIX & TIX′ = new

select? f: Fare
context BAL = 0
effect FARE′ = f

incoin? c: Coin
context BAL < nat (FARE)
effect BAL′ = BAL + int (c) & COINS′ = insert (COIN, c)

deliver! f: Fare
context BAL $\geq$ nat (FARE)
effect BAL′ = BAL − nat (FARE) & f = FARE & TIX′ = insert (TIX, f)

outcoin! c: Coin
context outcoin-poss
effect BAL′ = BAL − nat (c) & nat (c) $\leq$ BAL & count (COINS, c) > 0

Figure 4.1: Specification of a ticket machine system

The light operation of the previous example (that indicated when the machine was ready to accept coins) has been omitted.

4.1.3 The Specification of AUDIT

The process AUDIT is trivial. It simply performs a sequence of report operations, their execution being constrained by the context of the report operation of TM_d. In a more realistic specification, AUDIT might be linked to a message stream data object by an operation for sending a sales report.

4.1.4 An Auxiliary Specification for TM_d

In the ticket machine specification of Chapter 3, we gave the types of the state components and operation arguments informally, and assumed the definition of their operators. For example, we wrote

$$\{10, 20, 50, 100\}$$

for the type of incoin's argument, and

$$B' = B + c$$

for its effect clause. We were relying on the intuition that the coin value type is an enumeration, and that coin values can be added to balances as if they were natural numbers. We now formalize this part of the specification. In the present example, we have defined the balance to be a natural, given the enumeration types names, and used the nat operator to coerce coin values into naturals. We have also included more complex types: the counts of coins and tickets are described using the multiset types Coin-Mset and Fare-Mset. (For simplicity, coins returned as change are now of the same denomination as inserted coins.)

The definitions of these types and their operators are given in a separate, auxiliary specification. The specification is written in the Larch Shared Language [Guttag 85b]. It consists of a number of *traits*. For each data specification, there is a *root trait* with the same name that defines the types[2] and operators it uses. Each root trait lies at the top of a hierarchy of traits, combined by textual inclusion.

The traits for TM_d are shown in Figure 4.2. Consider the Coin-Mset trait first. The **imports** is a textual inclusion mechanism whose details need not concern us here; it brings in the axioms for the types Nat (natural numbers) and Coin. The type signatures of four operators are then given in the **introduces** section. (The first is a nullary operator, and may be thought of as a constant.) The equations following the **asserts** define the operators by relating them to each other. The first says that the new multiset has a size of zero. The second says that inserting an element increases the size by one. The third

[2]Guttag et al. talk of *sorts*, instead of *types*, in algebraic specifications, to distinguish them from their programming language counterparts. Since the types of our data specification correspond one-to-one with the sorts of our algebraic specification, and we want to avoid extra terminology, we do not distinguish them here.

TM_d: **trait**
imports Coin-Mset, Fare-Mset, Coin, Fare
introduces
possChangeAmt: Nat → Bool
asserts for all [x: Nat]
possChangeAmt (x) ==
(x=10) | (x=20) | (x=30) | (x=40) | (x=50) | (x=60) | (x=70) | (x=80) | (x=90)

Coin-Mset: **trait**
imports Natural, Coin
introduces
new: Coin-Mset
insert: Coin-Mset, Coin → Nat
size: Coin-Mset → Nat
count: Coin-Mset, E → Nat
total: Coin-Mset → Nat
#⊆#: Coin-Mset, Coin-Mset → Bool
asserts
Coin-Mset **generated by** [new, insert]
Coin-Mset **partitioned by** [count]
for all [c, c_1, c_2 : Coin, m, m_1, m_2, m_3: Coin-Mset]
size (new) == 0
size (insert (m, c)) == size (m) + 1
count (new, c) == 0
count (insert (m, c_1), c_2) == count (m, c_2) + (**if** $c_1 = c_2$ **then** 1 **else** 0)
total (new) == 0
total (insert (m, c_1)) == total (m) + nat (c_1)
new ⊆ m
insert (m_1, c) ⊆ m_2 == m_1 ⊆ m_2 & count (m_2, c) > count (m_1, c)

Coin: **trait**
imports Natural
introduces
Coin **enumeration of** [10p, 20p, 50p, £1]
nat: Coin → Nat
asserts
nat (10p) == 10
nat (20p) == 20
nat (50p) == 50
nat (£1) == 100

Figure 4.2: Auxiliary specifications for TM_d

and fourth are the corresponding axioms for the count operator; the fourth tells us that inserting an element increments by one the count of the elements of that value. The remaining axioms define the total and $\subseteq$ operators.

The **partitioned by** indicates that count alone is enough to distinguish unequal terms of type Coin-Mset. The **generated by** adds an inductive rule of inference that can be used to prove properties that are true of all terms of type Coin-Mset; it says that each term that does not contain any variables of type Coin-Mset is equal to some term in which new and insert are the only operators with range Coin-Mset.

The trait for Coin is simpler. The enumeration shorthand defines 10p, 20p, 50p and £1 as nullary operators that generate Coin. A coercion operator nat that converts denominations to natural numbers (equal to their value in pence) is defined.

The types Fare-Mset and Fare have traits similar to Coin-Mset and Coin (but are not included here). Fare has only two, nullary operators, 60 and 80, for the two fare values. No other operators are needed (except equality, whose theory is automatically included) since we do no computation on fares.[3]

All these traits are brought together by textual inclusion in the root trait TM_d. The operator possChangeAmt, which does not fit logically into the other traits, is defined here.

The Auxiliary Specification's Contribution to the Semantics

When we gave the semantics of data specification in Section 3.4.5, we did not explain how we determine whether a context or effect holds. The auxiliary specification helps here. First, we interpret the auxiliary specification as a set of first-order formulae, called its theory. We then take the context or effect clause in question, and replace the unbound state variables by the algebraic terms denoting their current values and the argument variable, if any, by its value, obtaining a formula without free variables. If the formula is in the theory, the clause holds, and we apply the relevant inference rule.

[3]To simplify our presentation, we have not used the re-usable traits provided by the Larch Handbook [Guttag 85a], nor taken advantage of Larch's inclusion and renaming mechanisms. In practice, the types Fare-Mset and Coin-Mset would be based on the generic type Multiset given in the handbook.

4.2 The Implementation

We now describe a CLU [Liskov 81] implementation of the specification above. It is not a practical implementation. However, it is a starting point for our discussion of wider issues in implementation. Also, it may shed light on the meaning of satisfaction and of the operational semantics of the specification.

The overall structure is as follows. Finite state machines (coded as circular data structures) represent the processes. The data object is coded as a cluster of procedures with a local, mutable state. There are two procedures for each operation of the specification: a procedure without side-effects that evaluates the context condition, and a procedure that models the execution of the operation itself. We simulate concurrent execution of the finite state machines by alternately giving each an opportunity to perform an operation. At each opportunity, we evaluate the context of each operation's data counterpart; if none of the contexts are true, no operation is performed, otherwise we perform the first operation whose context is evaluated as true.

4.2.1 Implementing The Data Object

CLU provides a program unit called a *cluster* especially for data abstractions, whose typing conventions prevent access to the representation of the data type from outside. We have implemented the data object as a cluster. The header of the cluster declares the procedures it exports:

```
TMd = cluster is
      setlight_context, setlight_op,
      select_context, select_op,
      incoin_context, incoin_op,
      deliver_context, deliver_op,
      outcoin_context, outcoin_op,
      report_context, report_op
```

The state of the data object is local to the cluster, and its type is the abstract type of the cluster. So we declare its representation to be the 'rep' type

```
rep = record
        [BAL: int,
        FARE: Fare,
        C10: int,
        C20: int,
        C50: int,
        C100: int,
        T60: int,
        T80: int]
```

and introduce the state as the own variable tm, with its initialization

```
own tm:rep := rep$
              {BAL: 0, FARE: 60,
              C10: 0, C20: 0, C50: 0, C100: 0,
              T60: 0, T80: 0}
```

The four 'C' components of the record implement COIN; each keeps a count of the number of coins of a particular denomination[4]. Likewise, the 'T' components implement TIX.

We shall look at the implementation of one of the operations, outcoin. It is implemented as two procedures. The first evaluates the context condition, and the second models the actual execution of the operation:

```
outcoin_context = proc () returns (bool)
  return (tm.BAL ≥ 10 & tm.C10 > 0 |
      tm.BAL ≥ 20 & tm.C20 > 0 |
      tm.BAL ≥ 50 & tm.C50 > 0 |
      tm.BAL ≥ 100 & tm.C100 > 0)
  end outcoin_context

outcoin_op = proc () returns (Coin)
  c: int
  if tm.BAL ≥ 100 & tm.C100 > 0 then
    c := 100
    tm.C100 := tm.C100 − 1
  elseif tm.BAL ≥ 50 & tm.C50 > 0 then
    c := 50
    tm.C50 := tm.C50 − 1
  elseif tm.BAL ≥ 20 & tm.C20 > 0 then
    c := 20
    tm.C20 := tm.C20 − 1
  elseif tm.BAL ≥ 10 & tm.C10 > 0 then
    c := 10
    tm.C10 := tm.C10 − 1
    end
  tm.BAL := tm.BAL − c
  return (c)
  end outcoin_op
```

Recall the specification of the operation (with the state macro expanded)

```
outcoin! c: Coin
  context ∃ c . count (COINS, c) > 0 & nat (c) ≤ B
  effect B′ = B − nat (c) & nat (c) ≤ B & count (COINS, c) > 0
```

[4]To show that this is a faithful representation of the abstract state component, we would provide a mapping (called an *abstraction function*) from the concrete values of the representation to the abstract values. The mapping must be a surjection; that is, there must be a concrete value corresponding to each abstract value. Some of the concrete values have no meaning as abstract values, so we restrict the domain of the abstraction function with a *representation invariant*. The subject of data abstraction is extensively treated in [Liskov 86]. Chapter 4 in particular explains the relationship between an abstract type and its representation. In our example, the relationships are simpler than usual: there is an isomorphism between the members of the representation meeting the invariant and the abstract values.

The procedure outcoin_context is a straightforward implementation of the context condition: it just evaluates the predicate for the four values c can take. An interesting question about modularity arises here. The process object TM_p constrains the order of operations so that deliver always precedes outcoin, and consequently, the maximum amount due in change is 90p. We might be tempted, therefore, to omit the last term of the disjunction, since we know that it will never be true that tm.BAL $\geq$ 100. This would be a mistake. The implementation of TM_d must satisfy its specification, which allows orderings of operations forbidden by TM_p.

The procedure outcoin_op implements the effect clause of the operation. We have resolved the non-determinism of the specification; instead of returning any available coin that is worth less than the balance, we have chosen to return the largest. So change is always returned in as few coins as possible, with the larger denominations delivered first. Note that the output argument of the operation is the returned result of the procedure, and that the procedure modifies the state component tm.BAL. The state of the data object is hidden both by CLU's type-checking system, which prevents access to the rep of a cluster by external procedures, and by our keeping the state internal to the cluster – it does not appear as an argument or result of any of the cluster's procedures.

4.2.2 Implementing The Process Objects

We have implemented two abstract types, regular expressions and finite state machines, to make programming the process objects easy. The timetable of each process object is written directly in CLU code as a string, and is converted to a regular expression by the procedure re$parse. The assignment to TMp_re of type re is written

```
TMp_re: re := re$parse ("(setlight select incoin+ deliver outcoin*)*")
```

A procedure of the finite state machine cluster, fsm$re_to_fsm, converts the regular expression to a circular data structure representing a machine equivalent to the expression:

```
TMp_fsm: fsm := fsm$re_to_fsm (TMp_re)
```

The process object AUDIT is coded as a finite state machine likewise.

4.2.3 Fitting The Objects Together

Having built the objects, we now need some mechanism to execute the system. The procedure try takes the name of a data object operation as its argument, and attempts to perform that operation. It first tests the context of the operation, and if it is true, it executes the operation. For example, the fragment dealing with outcoin is

```
    ...
  if TMd$outcoin_context () then
    c: Coin := TMd$outcoin_op ()
    formatd_write ("Out pops coin ", int$unparse (c))
    signal executed end
    ...
```

The call to formatd_write has been inserted so we can trace the behaviour of the system; it displays the value of the returned coin on the screen.

The procedure attempt_fsm_step takes a finite state machine as its argument, and tries each operation allowed as the next transition of the machine once, by calling try. If the context of the operation tried is true, try signals 'executed' and attempt_fsm_step returns. If the execution of no operation succeeds, attempt_fsm_step returns normally.

```
attempt_fsm_step = proc (f: fsm) returns (fsm)
  for op: Op_name in fsm$next_op (f) do
    try (op)
      except when executed: f:= fsm$step (f, op) break end
    end
  return (f)
  end attempt_fsm_step
```

We simulate the parallel execution of the two processes by giving each alternately an opportunity to perform one operation, terminating when both machines have reached states from which there are no transitions:

```
simulate_conc = proc (f1, f2: fsm)
  while ~fsm$empty (f1) | ~fsm$empty (f2) do
    if ~fsm$empty (f1) then
      f1 := attempt_fsm_step (f1)
      end
    if ~fsm$empty (f2) then
      f2 := attempt_fsm_step (f2)
      end
    end
  end simulate_conc
```

Interleaving the executions of the two processes relies on the assumption that they do not share operations. The ticket machine system is executed by the statement

```
simulate_conc (TMp_fsm, AUDIT_fsm)
```

The example was coded to interact with a keyboard and screen. We have not expressed these as objects, as that seemed an unnecessary complication here. Instead, the input arguments should be thought of as being chosen non-deterministically; the mechanism supplying the values (namely, keyboard interaction) is outside the scope of the specification.

The entire code is given in Appendix A, and a transcript of a sample interaction in Appendix B.

4.3 Discussion

The implementation scheme we have outlined above depends on many simplifying assumptions. We now consider each assumption in turn, and suggest some of the complications that would arise if it were relaxed.

4.3.1 Sharing of Operations

We assumed that the process objects did not share operations. Had they done so, we would not have been able to execute the parallel combination by interleaving, since the objects would have had to synchronize on common operations. Furthermore, since we had only one data object, there was no question of operations belonging to more than one data object. These are not significant complications. We would elaborate the try procedure so that it tests the contexts of the common operation in all its data objects. Then, if all contexts are true, the operation is performed in each, with inputs and outputs matched appropriately.

4.3.2 Scheduling

By running the processes sequentially, we avoided a potential hazard. Between evaluation of an operation's context and its execution, the state of the data object might change. In a parallel execution, therefore, we would have to ensure that the testing of the context and the execution of the operation is a single, indivisible transaction. A lock would probably be the simplest way to achieve this. Before evaluating the context, a 'mutex' is seized, preventing access by another party. After the execution of the operation (or after the context test if it returned false), the mutex is released. If operations were shared between data objects, then *all* of the context tests and executions of a shared operation would have to occur within a transaction.

It will often be useful to run concurrent processes sequentially, as we have done in our example. A practical implementation scheme would offer (and justify) one or more mechanisms for 'sequentializing' parallel combinations. Inversion in JSD [Jackson 83] is an example of such a mechanism, in which processes are run as coroutines, transferring control on message stream reads and writes.

The scheduling will generally depend both on the hardware we have available and on fairness (see Section 3.6.1) and liveness constraints that have not been formally specified. A liveness criterion might require, for example, that the user of a terminal is never kept waiting for more than three seconds, or might limit average response time. Although fairness can often be formulated as a safety property, we might favour a simpler specification, leaving fairness concerns to later in development. In the ticket machine system, for example, we assumed that report occurs 'often enough'. A correct (but unreasonable) implementation of the ticket machine system above would never execute the report operation of AUDIT at all. There are many other schedulings that might have been acceptable. We could have a fixed interleaving of the two processes, attempting a report operation in AUDIT after every deliver operation of TM_p, say, or we could use a clock to time the AUDIT process, attempting report once an hour, or once a day.

4.3.3 Renaming and Database Design

Implementing the state of a data object as the local state of a cluster suffices here because we only have one data object. A real system is likely to have hundreds or even millions of data objects, most being instantiations, by renaming, of generic objects. A banking system, for example, could have an account data object for each bank account. Allocating the storage of the objects' states and building accessing mechanisms is the subject of database design. We have not addressed the question of how a database design would satisfy part of our specification, let alone how such a design would be arrived at.

4.3.4 Conclusions

Although our implementation is deficient in many respects, and the style is not well-developed, we do believe that it is promising. It is encouraging that the implementation respects the modularity of the specification – the encapsulation of the operations of the data object in particular. We are also pleased with the direct execution of the process timetables. The separation of process and data clarifies many of the implementation issues we have raised. Scheduling issues can be considered for the process objects alone, taking advantage of their simple expression of the gross sequencing. Similarly, database design will be concerned mainly with the data objects. Not all the restrictions we have assumed are unrealistic; it may, for instance, be acceptable to forbid the sharing of operations between process objects, so that they communicate only via data objects.

Chapter 5

Discussion & Summary

5.1 Discussion

5.1.1 Specifying The Structure of the Implementation

We pointed out, in our treatment of satisfaction (in Section 3.5), that the satisfaction ordering on semantic objects is only the first step towards a practical implementation scheme. We have a much more restrictive notion of satisfaction in mind.

A specification should impose more than behavioural constraints on an implementation. This is not a new suggestion: advocates of the data approach have always taken it for granted that the operations of an abstract data type are to be implemented as separate code fragments. The structure of a specification is not determined merely by how easy it is to understand; it must influence the structure of the implementation too.

Modularity & Maintenance

A software system should be *modular*. Division into modules does not, by itself, make a system modular. The system must *maintain* the division over time; it must be possible to accommodate changes within modules, so that their effect is localized. Thus, we can only judge whether a system is modular if we know what changes are likely to be made. A classic example is provided by the data approach. The representation of an abstract data type is likely to change, so we encapsulate it within a single module, hiding it from the rest of the system.

In general, we need to study the subject matter of the system to know which modifications are most probable, and then we can design the system with those modifications in mind. Fortunately, it is rarely necessary to predict exactly which modifications will be demanded. The designer of a data abstraction knows that changes in representation are likely, but not which changes. The first stage in the JSD method [Jackson 83] is an explicit modelling of the subject matter of the system. Jackson contends that this makes the system more maintainable; the set of functions that may be added to the system is no longer unpredictable, but depends on the richness of the model. Good designers are aware of this principle (at least subconsciously): it is clear, for example, that a banking system must have a component representing an account, whether it is a database record,

a data abstraction, a process or whatever. Explicit modelling is also justified as a formalization of the client's understanding of the problem domain upon which a more rigorous description of requirements may be based [Cameron 86].

Specifying Internal Structure

This view of maintenance emphasizes the structure of the specification more than traditional approaches. The specification aphorism 'describe *what* and not *how*' is commonly interpreted as a condemnation of specifications that constrain the internal structure of the specification. We believe that, to the contrary, if the structure of an implementation is not constrained by its specification, it is likely to be unmodifiable.

We expect implementations of our specifications to respect most of their structure. Implementation will be concerned mainly with resource allocation: scheduling the available processors amongst the objects and designing data structures and algorithms to manipulate them efficiently. The implementation structure must correspond to the specification structure both in the division into objects, and in the internal structure of the objects. We do not think that it will be easy to formalize this notion of satisfaction, and it is likely to remain an informal appendage to the formal satisfaction notion for some time.

Data Objects

The implementation must enforce the encapsulation of the operations. Also, the state of the data object must be implemented. This is controversial; some advocates of the data approach insist that it is the behaviour of the operations only that matters, and that a specification that employs a state containing more information than strictly necessary is *biased.* This is a common complaint against model-oriented specifications: specifiers may over-constrain an implementation by demanding properties of the state that are inessential. Suppose, for example, that we based the specification of a set on a sequence. We specify an insert operation that inserts an element by appending it to the end of the sequence, a member operation that tests whether an element is in the sequence and a choose operation that removes an arbitrary element from the set. We might think that if we allowed any element of the sequence to be returned the specification would not be biased. It is true that we have not over-constrained the *behaviour* of the specification, since we cannot determine the order in which elements were inserted by using choose or member. However, we would have over-constrained the *state.* The usual notion of satisfaction is not applied to the behaviour of the operations in relation to one another, but to the state transformations of each operation. Since insert appended the new element to the *end* of the sequence, the implementation must order the elements too. Jones offers a criterion for implementation bias [Jones 86]: if two values of the state of the specification can be distinguished by the primitive operators of its type, but not by the operations of the specified type, then the specification is biased.

The criticism brought against model-oriented specifications is less relevant to our specifications. A data specification's state is not merely a device to help describe its behaviour: it is usually embodies genuine state in the problem domain. We construct

the state from primitive types; these will not be biased since we can define them by their properties alone in an auxiliary specification (as shown in Section 4.1.4). We argued in Chapter 2 that data specifications are suitable only when the state is important. When the state is unimportant, we would expect to use a process object, and then, since the specification does not mention state, the question of bias does not arise. One purpose of a specification is precisely to bias the implementation; our method allows bias to be used where appropriate and avoided elsewhere.

Behavioural implementations may be unmaintainable. Consider a pocket calculator offering statistical functions[1]. Suppose we specified a data object with operations enter, clear, mean and sd. We choose a multiset as the state; enter inserts elements into the multiset an clear empties it. mean and sd are defined according to their textbook definitions: the mean is the sum of all the elements divided by the number of elements, and the standard deviation is the root of the sum of the squares of the deviations from the mean. Now notice that this specification is 'biased'. The mean and standard deviation functions can be defined if the state retains less information. All we need keep is the number of values entered, their sum and the sum of their squares. So we might implement the data object with this slimmer state, so that the implementation's behaviour, but not its structure, satisfied the specification. Suppose then that we wanted to add a median function to the calculator: we would have to elaborate the state. The multiset was not a technical device. We think of statistical samples as sequences of values whose order does not matter, and the implementation ignored this. Of course, one can still pick an inappropriate state: for some statistical applications the order of elements in the sample does matter. (Some pocket calculators interpret a sequence of elements entered in order as a set of pairs, providing the first element of each pair automatically by incrementing a counter.)

Process Objects

We also intend that the structure of the process specification be evident in the implementation. A reasonable elaboration of the process notation would allow the timetable to be expressed as a regular grammar rather than a regular expression. This allows us to introduce names for non-terminals that may be important in maintenance. For example, use of a workstation might be described by

WKS $\widehat{=}$ **Operations are**
user?: Name
passwd?: Pass
no?, yes?, logout?

Timetable is SESSION*
SESSION = BADSESSION $\Box$ GOODSESSION
GOODSESSION = GOODLOGIN logout?
BADSESSION = BADLOGIN$^+$ GOODLOGIN logout?
BADLOGIN = ENTRY no?

[1] Example due to Carroll Morgan

GOODLOGIN = ENTRY yes?
ENTRY = user? passwd?

The specifier considered the distinction between 'bad sessions' and 'good sessions' significant, perhaps because users who failed once to gain entry should be considered suspect throughout their session. Consider a different specification

WKS′ ≙ **Operations are**
user?: Name
passwd?: Pass
no?, yes?, logout?

Timetable is SESSION*
SESSION = BADLOGIN* GOODLOGIN logout?
BADLOGIN = ENTRY no?
GOODLOGIN = ENTRY yes?
ENTRY = user? passwd?

These specification describe the same semantic objects. The timetable grammars denote the same set of operation sentences. But the grammars are structured differently; the second makes no distinction between good and bad sessions. Suppose the system administrator wants a report listing the start and end times of all sessions that began with an incorrect password entry. Two new operations are to be inserted: one at the start of such a session, and one at the end. It is clear where to put them in WKS – at the start and end of the BADSESSION component. But the operations cannot be placed in WKS′ without restructuring. An implementation of WKS that did not respect the structure of the timetable would not bear this modification.

This idea is familiar to compiler writers. A reference grammar defines the language recognized by the compiler, but a more elaborate grammar – the working grammar – is used during development of the compiler. The working grammar makes more distinctions than the reference grammar, making it easier to introduce static semantic checks and code generation.

Specification of internal structure is common to the operational approaches. But there are some aspects of the implementation's structure that we must not constrain – for example, the internal structure of the operations of a data object. Zave explains the distinction to be between 'problem-oriented' structure and 'implementation-oriented' structure [Zave 84].

Other Satisfaction Notions

In some circumstances it may be desirable for an implementation *not* to satisfy the structural constraints of the specification. Sometimes, for example, maintenance is not a concern at all. Nobody would seriously suggest that an electronic calculator should be constructed with modification in mind. A disposable piece of software that is not to

be maintained might not have to satisfy structural constraints (but no real system will be disposable). Furthermore, most specifications have errors in them, so modifications may occur before the software is even complete. Implementers also make errors, and a well-structured implementation is much easier to work with.

Implementations generated by mechanical transformations should also be less constrained. Structural constraints are justified by maintenance; if we never intend to maintain a document by hand, its structure does not matter. Programmers are careful to make their programs understandable, but a compiler generates code to run fast, not to be readable. So, if there are useful transformations that could be used in implementing our specifications, we would not expect them to satisfy the same criteria as a manual implementation technique.

5.2 Related Work

5.2.1 Data & Process

We have not found any work that addresses the combination of data and process in a manner analagous to ours. Campbell & Habermann use regular expressions to specify the permissible operation sequences on a typed object, although their concern is limited to synchronization of access to shared resources [Campbell 74]. Bartussek & Parnas have developed a specification language in which separate axioms define the meaning of the operators and 'legal traces', the orders in which they may be invoked [Bartussek 77].

As we discussed in Chapter 2, most approaches to software development incorporate notions of both data and process, but invariably, one is dominant and the other insufficiently treated. Our experience has suggested that it may not be as easy as it seems to extend an existing approach. We started by trying to generalize the notion of indexing in CSP [Hoare 85]. An example from Hoare's book [Hoare 85] introduces the notion of indexing:

> An object starts on the ground, and may move *up*. At any time thereafter, it may move *up* or *down*, except that when on the ground it cannot move any further down. But when it is on the ground, it may move *around*. Let n range over the natural numbers {0,1,2,...}. For each n, introduce the indexed name CT_n to describe the behaviour of the object when it is n moves off the ground. Its initial behaviour is defined as
>
> $$CT_0 = (up \rightarrow CT_1 \mid around \rightarrow CT_0)$$
>
> and the remaining infinite set of equations consists of
>
> $$CT_{n+1} = (up \rightarrow CT_{n+2} \mid down \rightarrow CT_n)$$
>
> where n ranges over the natural numbers 0,1,2, ...

Later in the book a buffer is described by indexing the process name with the sequence of symbols 'stored'; the index no longer follows a simple progression as in a recurrence relation, but is manipulated by more general operators on sequences.

We considered extending this to indexing of process names with the values of abstract types, whose operators were specified by an accompanying 'data' specification. So, for example, the bank account of Section 2.2.3 became

$$\begin{aligned}
\mathrm{ACC} &= \mathrm{open?}\ d \rightarrow \mathrm{ACC}_{init(d)} \\
\mathrm{ACC}_A &= \mathrm{payin?}\ p \rightarrow \mathrm{ACC}_{deposit(A,p)} \\
&\quad \mid \mathrm{wdraw?}\ w \rightarrow \mathrm{ACC}_{withdraw(A,w)} \\
&\quad \mid \mathrm{bal!}\ balance(\mathrm{A}) \rightarrow \mathrm{ACC}_A \\
&\quad \mid \mathrm{close} \rightarrow \mathrm{STOP}
\end{aligned}$$

where A has abstract type *Account*, with operators (that is, side-effect free functions) *init*, *deposit*, *withdraw* and *balance* (defined by a conventional algebraic specification in Larch [Guttag 85b]).

This idea failed for two reasons:

1. We often wanted to impose conditions on event occurrences that did not fit comfortably in the process equations. Suppose, for example, we wanted to disallow **wdraw** events when the balance is below zero. There is no natural place to write this condition in the process equation. We could place bounds on the distributed choice operator preceding **wdraw**; this only works here because **wdraw** only appears once in the process expression. The condition is associated with the set of **wdraw** events: it is a condition on an *operation*, and there are no operations here.

2. We were unhappy interpreting the CSP equations as state machines. We wanted both to retain the trace semantics of CSP and at the same time to incorporate the notion of state transitions for the abstract state, but this was infeasible.

We had characterized the paradigms wrongly, postulating that the process part was about scheduling, and the data part was about values and relationships between values. Our data specifications had no temporal aspects; they described spaces of immutable values that would be passed around within the process specification. We later abandoned that view. The essence of the data part is *state* and the partitioning of events into operations. Our data specifications do incorporate scheduling constraints, but only those related to the data state, applied homogeneously to all the events of an operation.

5.2.2 Composing Descriptions

In [Jackson 86], Jackson proposes that software creation be viewed as the production and manipulation of descriptions. In the same way that the mechanical engineer is not restricted to a single material – a car cannot be made entirely out of steel, rubber or glass – so software developers should be able to combine different sorts of descriptions. Different languages are suitable to different aspects of a software system, and, rather than aspiring to languages of universal application, we should concentrate on combining existing languages. He argues that CSP's notion of parallel combination is more flexible than the 'whole-and-part' composition mechanism of procedure call.

Our work is an example of a composition of two specification languages, using a combinator akin to the CSP combinator. Zave has been working recently on combining implementation languages; an example in [Zave 88a] demonstrates the combination of CSP, Prolog and PAISLey in a telephone switching system. Her approach, called *view composition*, is described in [Zave 88b].

The Larch specification language [Guttag 85b] is a successful composition of two aspects of a data specification. Each specification is written in two *tiers*. The *interface language* tier describes procedures by means of predicates giving pre- and post-conditions. These predicates contain *operators* that are defined algebraically in the *Shared Language* tier. The two tiers make it possible to separate the definition of underlying mathematical abstractions from the description of language-dependent interface details, such as shared memory, side-effects and exception handling. There is one Larch interface language for each programming language.

5.3 Further Work

5.3.1 The Model

Non-Determinism

The treatment of non-determinism is deficient in our model. Consider first the notion of internal operations. Internal operations are used to give an object autonomy in the choice and execution of an operation. Unlike the hidden events of CSP, they do occur in traces. They are useful for describing mechanisms that are outside the scope of the specification. For example, the ticket machine may have a sensor that raises a signal when the ticket supply is low. We might represent the signalling by an internal operation in our specification, since, for the software, it occurs non-deterministically. Suppose however that we wanted to offer the implementer a choice of *external* operation at some point. This is not easily represented in our model. We can simulate a non-deterministic choice between the external operations by prefixing them with internal operations. For example, a ticket machine that may deliver pink or yellow tickets as it pleases may be specified as

$TM_p \triangleq$ **Operations are**
select?: {60,80}
incoin?: {10,20,50,100}
chooseY, chooseP
delY?: {60,80}
delP?: {60,80}
outcoin?: {10,20}

Timetable is (select? incoin?$^+$ (chooseP delP? $\Box$ chooseY delY?) outcoin?*)*

A trace of TM_p is

<select.60, incoin.50, incoin.10, chooseP, delP.60>

The chooseP operations is out of place, and we would rather it did not have to appear. Even worse, chooseP and chooseY must both be operations of the implementation object, so a ticket machine that only knew of yellow tickets would not be acceptable.

A proper treatment of non-determinism requires the notion of *refusals* or *ready sets* [Brookes 84]. Instead of considering the set of operations that can occur after some trace, we would have to consider a set of sets of operations. In one formulation, each set of operations is a 'ready' set, and represents the set of operations offered by the object after a sequence of one or more internal choices. The liveness satisfaction criterion would then require that any ready set of the implementation is a superset of every ready set of the specification.

Another deficiency of our model is the notion of crashing, which is extremely crude. Introducing full non-determinism would allow us to specify chaotic (*ie*, unpredictable) behaviour, and we could dispense with crashing.

Recall that our omission of full non-determinism from our model prevented us from allowing the context of a data operation to be non-deterministic. There are cases in which this would be desirable. Suppose our ticket machine sends a message to a central depot when the change supply is low. We might want to specify that it *must* be sent if the machine cannot guarantee change for the next transaction, but that it *may* be sent earlier, when the change level drops below some specified point.

New Combinators

It may be useful to add other combinators to the model. In describing shared resources an interleaving combinator is convenient. For example, we might have a message queue data object, written to by several distinct objects. We could give the writing operation of each of these a different name, and, by renaming (Section 3.6.5), make each of these a pseudonym of the write operation provided by the message queue. This is cumbersome, and it is also unfortunate that to add a new writing object we would have to amend the renaming since the shared resource must know the names of its clients. An interleaving combinator would solve this problem. By interleaving the writing objects, and linking (Section 3.3.2) that combination to the shared queue, we could enforce synchronization between each writer and the message queue while allowing the writers to run independently.

A hiding operator might also be useful. Its absence was evident in the queue merger example of Section 3.6.6 when it would have been appropriate to hide the internal operations. We have avoided it because it introduces more non-determinism than our model can handle. Having included general non-determinism, it would be worth adding a hiding operator too.

5.3.2 Specifications

Our specification language is too primitive for practical use. A syntax should be developed that is both easy to read and suitable for machine manipulation. For editing system

specifications it would be useful to elaborate the combinator and enclosure notation to include various redundancy checks (as in the Larch Shared Language [Guttag 85b]): the specifier could record, for instance, which operations are intended to link two objects, and which operations remain external after enclosure.

Process Objects

As we mentioned above (Section 5.1.1), we expect the timetable of a process object to be expressed as a regular grammar rather than a regular expression: the naming of non-terminals aids maintenance, as well as breaking the timetable into more comprehensible pieces. We might use a context-free grammar for its additional expressive power.

Data Objects

The notation for data objects could benefit from several elaborations. We gave an example of a data specification that used operators and types defined in an auxiliary algebraic specification (Section 4.1.4) but we did not give an adequate syntax for incorporating the auxiliary specifications. Constructing the type of the state with domain equations from primitive types (as in VDM) may be useful too. A 'modifies at most' clause, as in the Larch CLU interface language, could be used to indicate which components of the state an operation may modify. The pre-condition of an input operation is only implicit in its effect clause; we could separate the effect clause into a pre-condition (on input arguments) and a post-condition.

5.4 Summary

To conclude, we give a brief summary of our work. We started by analyzing two specifications of the same system, one written in the data paradigm and one in the process paradigm. The data specification defined a state, and a set of operations that modified the state. The process specification constrained the sequencing of events. We considered some simple elaborations of the specification, and saw that each was easy to express in only one of the specifications. We postulated that *state* and *sequencing* are the essences of the data and process paradigms respectively, and we argued that omission of either of these aspects has dire consequences for system development. Without explicit data, a system cannot take advantage of representation independence, a vital tool for reducing complexity. Without sequencing notions, the components of a system cannot be connected in an abstract fashion.

An informal specification of a ticket machine fell naturally into two parts. We presented a formal specification mirroring the two parts; the data part defined a state and operations on the state and the process part defined an explicit ordering on the operations. The data part restricted the order of operations, by giving a context for each operation. We gave a formal semantics of our specification in terms of traces. Each part of the specification denoted an object; the meaning of the composite specification was

Our specification language is too primitive for practical use. A syntax should be developed that is both easy to read and suitable for machine manipulation. For editing system

Our formal semantics can handle arbitrary combinations of objects. In a series of small examples, we showed how to combine several process and data objects. We added two new operators on objects, renaming and enclosure, that are useful in building systems.

We defined a satisfaction ordering on objects — the first step towards an implementation technique. In a small case study, we presented a more elaborate form of the data specification that incorporated an algebraic specification. We implemented it in CLU, and discussed some of the issues that a practical implementation scheme would have to address. Finally, we outlined a prescriptive notion of implementation in which the specification imposes constraints on the structure of the implementation.

The contributions of our work are two-fold. First, we hope that our analysis of the data and process approaches has clarified the differences between them, and will lead to a better appreciation of their strengths and weaknesses. Secondly, we have shown how to combine data and process in a software specification.

Appendix A

Ticket Machine Code

```
Op_name = string  %Global type definitions: CLU has no enumeration type,
Coin = int        %so Coin and Fare are defined as ints, which is unsafe,
Fare = int        %but adequate here.

start_up = proc ()
   %make a ticket machine process object
   TMp_re: re := re$parse ("(setlight select incoin+ deliver outcoin*)*")
   TMp_fsm: fsm := fsm$re_to_fsm (TMp_re)

   %make auditing process
   AUDIT_re: re := re$parse ("report")
   AUDIT_fsm: fsm := fsm$re_to_fsm (AUDIT_re)

   %simulate concurrent execution of ticket machine and auditor
   simulate_conc (TMp_fsm, AUDIT_fsm)
   end start_up

simulate_conc = proc (f1: fsm, f2: fsm)
   while ~fsm$empty (f1) | ~fsm$empty (f2) do
      if ~fsm$empty (f1) then
         f1 := attempt_fsm_step (f1)
         end
      if ~fsm$empty (f2) then
         f2 := attempt_fsm_step (f2)
         end
      end
   end simulate_conc

attempt_fsm_step = proc (f: fsm) returns (fsm)
   for op: Op_name in fsm$next_op (f) do
      try (op) except when executed: f:= fsm$step (f, op) break end
      end
   return (f)
   end attempt_fsm_step
```

```
try = proc (op: Op_name) signals (executed)
  if op = "setlight" then
    if TMd$setlight_context () then
      b: bool := TMd$setlight_op ()
      if b then
        write ("Change available")
      else
        write ("Out of change")
        end
      signal executed
      end
  elseif op = "select" then
    if TMd$select_context () then
      f: Fare := int$parse (prompt_read ("Select fare" ))
      TMd$select_op (f)
      signal executed
      end
  elseif op = "incoin" then
    if TMd$incoin_context () then
      c: Coin := int$parse (prompt_read ("Insert coin "))
      TMd$incoin_op (c)
      signal executed
      end
  elseif op = "deliver" then
    if TMd$deliver_context () then
      f: Fare := TMd$deliver_op ()
      formatd_write ("Out pops ticket of fare ", int$unparse (f))
      signal executed
      end
  elseif op = "outcoin" then
    if TMd$outcoin_context () then
      c: Coin := TMd$outcoin_op ()
      formatd_write ("Out pops coin ", int$unparse (c))
      signal executed
      end
  elseif op = "report" then
    if TMd$report_context () then
      c1, c2: int := TMd$report_op ()
      formatd_write ("Sales of 60p tickets ", int$unparse (c1))
      formatd_write ("Sales of 80p tickets ", int$unparse (c2))
      signal executed
      end
    end
  end try
```

```
TMd = cluster is
  %Ticket Machine Data Object Cluster
    setlight_context, setlight_op,
    select_context, select_op,
    incoin_context, incoin_op,
    deliver_context, deliver_op,
    outcoin_context, outcoin_op,
    report_context, report_op

  rep = record
        [BAL: int,
        FARE: Fare,
        C10: int,
        C20: int,
        C50: int,
        C100: int,
        T60: int,
        T80: int]

  own tm: rep := rep$
              {BAL: 0, FARE: 60,
              C10: 0, C20: 0, C50: 0, C100: 0,
              T60: 0, T80: 0}

  report_context = proc () returns (bool)
    return (tm.T60 + tm.T80 > 5)
    end report_context

  report_op = proc () returns (int, int)
    sales_60: int := tm.T60
    sales_80: int := tm.T80
    tm.T60 := 0
    tm.T80 := 0
    return (sales_60, sales_80)
    end report_op

  setlight_context = proc () returns (bool)
    return (~ outcoin_context ())
    end setlight_context

  setlight_op = proc () returns (bool)
    return (tm.C10 ≥ 9)
    end setlight_op
```

```
select_context = proc () returns (bool)
   return (true)
   end select_context

select_op = proc (f: Fare)
   tm.FARE := f
   end select_op

incoin_context = proc () returns (bool)
   return (tm.BAL < tm.FARE)
   end incoin_context

incoin_op = proc (c: Coin)
   tm.BAL := tm.BAL + c
   if c = 100 then
      tm.C100 := tm.C100 + 1
   elseif c = 50 then
      tm.C50 := tm.C50 + 1
   elseif c = 20 then
      tm.C20 := tm.C20 + 1
   elseif c = 10 then
      tm.C10 := tm.C10 + 1
      end
   end incoin_op

deliver_context = proc () returns (bool)
   return (tm.BAL ≥ tm.FARE)
   end deliver_context

deliver_op = proc () returns (Fare)
   if tm.FARE = 60 then
      tm.T60 := tm.T60 + 1
   elseif tm.FARE = 80 then
      tm.T80 := tm.T80 + 1
      end
   tm.BAL := tm.BAL − tm.FARE
   return (tm.FARE)
   end deliver_op

outcoin_context = proc () returns (bool)
   return (tm.BAL ≥ 10 & tm.C10 > 0 |
        tm.BAL ≥ 20 & tm.C20 > 0 |
        tm.BAL ≥ 50 & tm.C50 > 0 |
        tm.BAL ≥ 100 & tm.C100 > 0)
   end outcoin_context
```

```
    outcoin_op = proc () returns (Coin)
      c: int
      if tm.BAL ≥ 100 & tm.C100 > 0 then
        c := 100
        tm.C100 := tm.C100 − 1
      elseif tm.BAL ≥ 50 & tm.C50 > 0 then
        c := 50
        tm.C50 := tm.C50 − 1
      elseif tm.BAL ≥ 20 & tm.C20 > 0 then
        c := 20
        tm.C20 := tm.C20 − 1
      elseif tm.BAL ≥ 10 & tm.C10 > 0 then
        c := 10
        tm.C10 := tm.C10 − 1
        end
      tm.BAL := tm.BAL − c
      return (c)
      end outcoin_op
    end TMd


re = cluster is parse, mk_tml, mk_choice, mk_iter, mk_seq,
            is_tml, is_seq, is_sel, is_itr,
            get_one, get_two, get_tml, get_body
    rep = oneof [tml: Op_name,
            seq: re_pair,
            sel: re_pair,
            itr: re]


    re_pair = record [one: re, two: re]

    parse = proc (s: string) returns (cvt)
      %parses the string s, and calls the procedures
      %mk_tml, mk_choice, mk_iter and mk_sel to construct
      %a regular expression.

    mk_tml = proc (op: Op_name) returns (cvt)
      return (rep$make_tml (op))
      end mk_tml

    mk_choice = proc (r1,r2: re) returns (cvt)
      return (rep$make_sel (re_pair${one: r1, two: r2}))
      end mk_choice
```

```
mk_iter = proc (r: re) returns (cvt)
   return (rep$make_itr (r))
   end mk_iter

mk_seq = proc (r1,r2: re) returns (cvt)
   return (rep$make_seq (re_pair${one: r1, two: r2}))
   end mk_seq

is_tml = proc (r: cvt) returns (bool)
   return (rep$is_tml (r))
   end is_tml

is_seq = proc (r: cvt) returns (bool)
   return (rep$is_seq (r))
   end is_seq

is_sel = proc (r: cvt) returns (bool)
   return (rep$is_sel (r))
   end is_sel

is_itr = proc (r: cvt) returns (bool)
   return (rep$is_itr (r))
   end is_itr

get_one = proc (r: cvt) returns (re) signals (not_seq_or_sel)
   %returns the first regular expression in a sequence or selection
   tagcase r
   tag sel (rp: re_pair): return (rp.one)
   tag seq (rp: re_pair): return (rp.one)
   others: signal not_seq_or_sel
   end
   end get_one

get_two = proc (r: cvt) returns (re) signals (not_seq_or_sel)
   %returns the second regular expression in a sequence or selection
   tagcase r
   tag sel (rp: re_pair): return (rp.two)
   tag seq (rp: re_pair): return (rp.two)
   others: signal not_seq_or_sel
   end
   end get_two

get_body = proc (r: cvt) returns (re) signals (not_itr)
   %returns the regular expression that is the body of an iteration
   tagcase r
   tag itr (r1: re): return (r1)
   others: signal not_itr
```

```
      end
      end get_body

   get_tml = proc (r: cvt) returns (Op_name) signals (not_tml)
      %returns the op name in a regular expression that is a terminal
      tagcase r
      tag tml (op: Op_name): return (op)
      others: signal not_tml
      end
      end get_tml
   end re


fsm = cluster is re_to_fsm, next_op, step, empty
   rep = record [init : state, final: state]
   state = array [edge]
   edge = record [labelled: bool, label: Op_name, dest: fsm]
   %An fsm is a non-deterministic finite automaton, represented
   %in the standard fashion. The procedure step executes
   %the fsm by re-assigning the pointer to the initial state.
   %Note that since only one initial state is maintained, when
   %two edges leaving some state have the same label, only one of
   %the transitions is taken. Regular expressions with 'backtracking'
   %are thus not properly handled.

   %The destination of an edge is an fsm, since making it a state
   %would violate Clu's rules for defining recursive types.

   next_op = iter (f: cvt) yields (Op_name)
      %returns a list of the operations that can occur next
      %according to the fsm f.
      for e: edge in state$elements (f.init) do
         if e.labelled then yield (e.label)
         else
            for op: Op_name in next_op (e.dest) do
               yield (op)
               end
            end
         end
      end next_op

   step = proc (f: cvt, op: Op_name) returns (fsm) signals (no_such_step)
      %returns the fsm obtained from f by performing the operation op
      for e: edge in state$elements (f.init) do
         if e.labelled then
            if e.label = op then return (e.dest) end
         else
```

```
            return (step (e.dest, op))
            end
        except when no_such_step: continue end
        end
    signal no_such_step
    end step

empty = proc (f: cvt) returns (bool)
    return (state$empty(f.init))
    end empty

new = proc () returns (cvt)
    %creates a new, empty finite state machine
    return (rep${init: state$[], final: state$[]})
    end new


re_to_fsm = proc (r: re) returns (cvt)
    %converts a regular expression into a finite state machine
    if re$is_tml (r) then
        %regular expression is a terminal
        op: Op_name := r.tml
        f: fsm := new ()
        si: state := state$[edge$ {labelled: true, label: op, dest: f}]
        return (rep${init: si, final: down (f).init})
    elseif re$is_seq (r) then
        %regular expression is a sequence
        f1: rep := down (re_to_fsm (r.one))
        f2: rep := down (re_to_fsm (r.two))
        %add an edge from the final state of f1 to the
        %initial state of f2
        e: edge := edge$ {labelled: false,
                  label: "",
                  dest: up (f2)}
        state$addh (f1.final, e)
        return (rep${init: f1.init, final: f2.final})
    elseif re$is_sel (r) then
        %regular expression is a selection
        f1: rep := down (re_to_fsm (r.one))
        f2: rep := down (re_to_fsm (r.two))
        %add a state with outgoing edges to the initial states of
        %f1 and f2
        si: state := state$[edge$ {labelled: false,
                        label: "",
                        dest: up (f1)},
                    edge$ {labelled: false,
                        label: "",
```

```
                        dest: up (f2)}]
        %add an edge from the final states of f1 and f2 to a
        %new dummy state
        f: fsm := new ()
        e1: edge := edge$ {labelled: false, label: "", dest: f}
        e2: edge := edge$ {labelled: false, label: "", dest: f}
        state$addh (f1.final, e1)
        state$addh (f2.final, e2)
        return (rep${init: si, final: down (f).init})
      elseif re$is_itr (r) then
        %regular expression is an iteration
        f: rep := down (re_to_fsm (r.body))
        %add a state with outgoing edges to the initial and final
        %states of the fsm representing r
        fb: rep := rep${init: f.final, final: f.final}
        si: state := state$[edge$ {labelled: false,
                        label: "",
                        dest: up (f)},
                    edge$ {labelled: false,
                        label: "",
                        dest: up (fb)}]
        %add an edge from the final state to the initial state
        e: edge := edge$ {labelled: false,
                label: "",
                dest: up (f)}
        state$addh (f.final, e)
        return (rep${init: si, final: f.final})
      end
      end re_to_fsm
    end fsm

write = proc (s: string)
    outs: stream := stream$primary_output ()
    stream$puts(outs, s)
    stream$putl(outs, "")
    end write

formatd_write = proc (s1, s2: string)
    outs: stream := stream$primary_output ()
    stream$puts(outs, s1)
    stream$putl(outs, s2)
    stream$putl(outs, "")
    end formatd_write
```

```
prompt_read = proc (s: string) returns (string)
   ins: stream := stream$primary_input ()
   outs: stream := stream$primary_output ()
   stream$puts(outs, s)
   entry: string := stream$getl(ins)
   stream$putl(outs, "")
   return(entry)
   end prompt_read
```

Appendix B

Transcript of Terminal Session

A transcript of a terminal session running the ticket machine follows. The text has been doctored slightly to improve formatting, and comments (in italics) have been added.

```
Script started on Fri Apr 29 11:35:50 1988
% a.out
Out of change                    Starts with no change & infinite ticket supply

Select fare 60                   1st customer inserts exact change
Insert coin 20
Insert coin 10
Insert coin 20
Insert coin 10
Out pops ticket of fare 60

Out of change                    ... because there are fewer than nine 10's

Select fare 80                   2nd customer inserts lots of 10's
Insert coin 10
Insert coin 10
Insert coin 10
Insert coin 10
Insert coin 10
Insert coin 10
Insert coin 10
Insert coin 10
Out pops ticket of fare 80

Change available                 Now machine has 'enough' change

Select fare 60                   3rd customer receives change
Insert coin 50
Insert coin 20
Out pops ticket of fare 60
Out pops coin 10
```

```
Change available

Select fare 60                          4th customer
Insert coin 50
Insert coin 100
Out pops ticket of fare 60
Out pops coin 50                        Larger denomination first
Out pops coin 20
Out pops coin 20

Change available

Select fare 60                          5th customer
Insert coin 50
Insert coin 100
Out pops ticket of fare 60
Out pops coin 50
Out pops coin 20
Out pops coin 10                        Machine has run out of 20's
Out pops coin 10

Out of change                           Now there are fewer than nine 10's

Select fare 60                          6th customer ignores warning, but gets change
Insert coin 50
Insert coin 100
Out pops ticket of fare 60

Sales of 60p tickets 5                  Auditing process reports ticket sales
Sales of 80p tickets 1

Out pops coin 50
Out pops coin 10
Out pops coin 10
Out pops coin 10
Out pops coin 10

Out of change

Select fare 60                          7th customer ignores warning & loses 10p
Insert coin 50
Insert coin 100
Out pops ticket of fare 60
Out pops coin 50
Out pops coin 10
Out pops coin 10
Out pops coin 10
```

```
Out of change                    ... but balance is 10p in favour of customer

Select fare 60                   8th customer saves 10p 'left' by 7th customer
Insert coin 50
Out pops ticket of fare 60

% exit
script done on Fri Apr 29 11:38:02 1988
```

Appendix C

Glossary

Regular Expressions

E^*	A sequence of zero or more E's.
E^+	A sequence of one or more E's.
$E \,[]\, F$	Choice of E or F.

Events and Traces

$op.a$	The event of operation op with argument a.
$\star$	The crash event or operation.
$<e, f>$	The trace consisting of the events e and f.
$<>$	The empty trace.
$s \cdot t$	The concatenation of trace s and trace t.
t^0	Trace t without its last event; $<>^0 = <>$.
t'	The last operation in trace t; $<>'$ is not defined.
T/t	The operations that may follow trace t in trace set T.
$T//t$	The events that may follow trace t in trace set T.
$t \mid_{OP}$	Trace t restricted to the events whose operations are in OP.
Seq(OP,A)	The set of legal event sequences for an object with operations OP and argument map A.

Objects

$X.Y$	Combination of objects X and Y.
$X \odot Y$	Combination (linking); X and Y share operations.
$X \oslash Y$	Combination (meshing); X and Y do not share operations.
$\Diamond_C X$	Enclosure of object X around output operations C.
$f(X)$	Renaming of object X with operation mapping f.
$i : X$	Object X labelled with i.
X[a,b for c]	Renaming of X; a and b are pseudonyms of c.
$X \sqsubseteq Y$	Satisfaction: object X is as good as object Y.

Bibliography

[Bartussek 77] W. Bartussek & D. Parnas, *Using Traces to Write Abstract Specifications for Software Modules*, TR 77-012, Dept. of CS, Univ. N. Carolina at Chapel Hill, 1977.

[Booch 86] G. Booch, 'Object Oriented Development', IEEE Trans. Softw. Eng., SE-12 (Feb 1986).

[Brookes 84] S.D. Brookes, C.A.R. Hoare & A.W. Roscoe, 'A Theory of Communicating Sequential Processes', Journal ACM 31 (7), 1984.

[Cameron 86] J.R. Cameron, 'An Overview of JSD', IEEE Trans. Softw. Eng., SE-12 (Feb 1986).

[Campbell 74] R.H Campbell & A.N. Habermann, 'The Specification of Process Synchronization by Path Expressions', Lecture Notes in CS, 16, Springer-Verlag, 1974.

[Cohen 86] B. Cohen, W.T. Harwood and M.I. Jackson, *The Specification of Complex Systems*, Addison-Wesley, 1986.

[Guttag 80] J.V. Guttag & J.J. Horning, 'Formal Specification as a Design Tool', Proc. Seventh ACM Symp. Principles Prog. Langs., Las Vegas, Nevada, Jan 1980.

[Guttag 85a] John Guttag, James Horning and Jeanette Wing, *Larch in Five Easy Pieces*, DEC Systems Research Center, TR-5 (1985).

[Guttag 85b] J. Guttag, J. Horning and J. Wing, 'The Larch Family of Specification Languages', IEEE Software, 2 (Sep 85).

[Hayes 86] I. Hayes, ed., *Specification Case Studies*, Prentice-Hall, 1986.

[Hoare 85] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice-Hall, 1985.

[Hopcroft 86] J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.

[Jackson 76] M.A. Jackson, 'Information Systems: Modelling, Sequencing and Transformations', Proc. 3rd Intl. Conf Softw. Eng., IEEE, 1978.

[Jackson 83] M.A. Jackson, *System Development*, Prentice-Hall 1983.

[Jackson 86] M.A. Jackson, *Keynote Address*, Proc. IFIP. 10th World Comp. Cong., Dublin 1986, IFIP, 1986.

[Jones 85] C.B. Jones, 'The Role of Proof Obligation in Software Design', Proc. TAPSOFT, Berlin 1985, Lecture Notes in CS, 186, Springer Verlag, 1985.

[Jones 86] C.B. Jones, *Systematic Software Development Using VDM*, Prentice-Hall, 1986.

[Liskov 81] B.H. Liskov, et al., *CLU Reference Manual*, Lecture Notes in CS, 114, Springer-Verlag, 1981.

[Liskov 86] B. Liskov and J. Guttag, *Abstraction & Specification in Program Development*, MIT Press, 1986.

[Meyer 85] B. Meyer, 'On Formalism in Specifications', IEEE Software, (Jan 1985).

[Olderog 86] E-R Olderog, 'Semantics of Concurrent Processes: The Search For Structure & Abstraction', Bulletin of EATCS (Feb 1986).

[Wirth 76] N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall 1976.

[Zave 82] P. Zave, 'An Operational Approach to Requirements Specification for Embedded Systems', IEEE Trans. Softw. Eng., SE-8 (May 1982)

[Zave 84] P. Zave, 'The Operational Versus The Conventional Approach to Software Development', Commun. ACM 27 (Feb 1984).

[Zave 88a] P. Zave, 'An Experiment in Composing Paradigms', submitted for publication.

[Zave 88b] P. Zave, 'Toward a Multiparadigm Prototyping Environment Based on View Composition', submitted for publication.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS

2a. SECURITY CLASSIFICATION AUTHORITY

2b. DECLASSIFICATION / DOWNGRADING SCHEDULE

3. DISTRIBUTION / AVAILABILITY OF REPORT
Approved for public release; distribution is unlimited.

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
MIT/LCS/TR-419

5. MONITORING ORGANIZATION REPORT NUMBER(S)
N00014-83-K-0125

6a. NAME OF PERFORMING ORGANIZATION
MIT Laboratory for Computer Science

6b. OFFICE SYMBOL (*If applicable*)

7a. NAME OF MONITORING ORGANIZATION
Office of Naval Research/Department of Navy

6c. ADDRESS (*City, State, and ZIP Code*)
545 Technology Square
Cambridge, MA 02139

7b. ADDRESS (*City, State, and ZIP Code*)
Information Systems Program
Arlington, VA 22217

8a. NAME OF FUNDING / SPONSORING ORGANIZATION
DARPA/DOD

8b. OFFICE SYMBOL (*If applicable*)

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER

8c. ADDRESS (*City, State, and ZIP Code*)
1400 Wilson Blvd.
Arlington, VA 22217

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (*Include Security Classification*)
Composing Data & Process Descriptions in the Design of Software Systems

12. PERSONAL AUTHOR(S)
Jackson, Daniel

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
Technical	FROM _____ TO _____	1988 May	86

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	Formal specification, programming methods, paradigms, software design, data abstraction, process models

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

Two paradigms are dominant in software development, the data paradigm and the process paradigm. Our contention is that relying exclusively on either is counter-productive.

In the data paradigm, a system is specified as operations acting on states. The process paradigm focuses on sequences of events. By analyzing two specifications for the same system, one in each paradigm, we show that the prime concerns of the approaches based on the data and process paradigms are state and sequencing respectively. Without explicit data, a system cannot take advantage of representation independence, a prerequisite of modularity; without sequencing notions, the components of a system cannot be connected in an abstract fashion. Fortunately, the qualities of the two paradigms are complementary, suggesting an approach that combines the two.

We present a framework in which data and process specifications are combined formally.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified

22a. NAME OF RESPONSIBLE INDIVIDUAL
Judy Little, Publications Coordinator

22b. TELEPHONE (*Include Area Code*)
(617) 253-5894

22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted.
All other editions are obsolete

☆U.S. Government Printing Office: 1985—507-047

19. A series of small examples shows how data and process specifications can be combined to build systems. We define formally what it means for an implementation to satisfy a specification, and we show a CLU implementation of one of our examples. Finally we outline a prescriptive notion of implementation in which the specification dictates the internal structure of the implementation.